Czech Technical University in Prague
Faculty of Information Technology
Department of Digital Design

Generation of High-Speed Network Device from High-Level Description

by

Ing. Pavel Benáček

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics

Prague, November 2016

Supervisor:
doc. Ing. Hana Kubátová, CSc.
Department of Digital Design
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Co-Supervisor:
Ing. Viktor Puš, Ph.D.
CESNET a.l.e.
Zikova 4
160 00 Prague 6
Czech Republic

Abstract

OpenFlow, as the most popular embodiment of Software-Defined Networking, provides a way to network dataplane configuration at runtime. The OpenFlow specification strictly defines a set of supported protocols and actions for further processing of incoming traffic (i.e., switches are still mostly fixed). However, modern requirements on networking hardware have a dynamic character and administrators of high-end networks want to react to new protocols, security threats, novel approaches in traffic engineering, and so on. This isn't feasible with static network hardware and leads to the need to replace the hardware more often than desired.

The aim of my dissertation thesis is to provide the process of mapping from abstract language to the architecture of network device which is suitable for automatic generation and capable to hit processing speed of 100 Gbps in single FPGA. The architecture of network device is based on predefined interfaces and modules which are connected to a high-speed processing pipeline. The text provides details of transformation process to individual blocks of network device: parser, deparser, Match+Action table, Match+Action router and Match+Action group. The text also demonstrates the usage of High-Level Synthesis tools to provide a custom processing engine which is capable to hit speed of 100 Gbps. Finally, the text provides three use cases for demonstration of flexibility and easy extensibility with new protocols and actions. Each use case was described in P4 language, translated to VHDL and tested in real hardware environment. The results show that generated devices are capable to meet throughput in range from 77.6 to 100 Gbps.

Keywords:

FPGA, High-Level Synthesis, Transformation, P4, SDN, High-Speed Computer Networks, 100 Gbps, VHSIC Hardware Description Language.

Acknowledgements

First of all, I would like to express my thanks to my dissertation thesis supervisors, Ing. Viktor Puš, Ph.D. and doc. Ing. Hana Kubátová, CSc., for their support, valuable comments and guidance. They have been a constant source of encouragement and insight during my research and helped me with numerous problems and professional advancements.

Special thanks go to the staff of the Liberouter project, funded by CESNET, and the Department of Digital Design of the Czech Technical University in Prague, who maintained a pleasant and flexible working environment for my research. I would like to express special thanks to the management of both institutions for providing of the funding for my research. My research has been partially supported by the Czech Technical University in Prague, grants No. SGS14/102/OHK3/042/14, No. SGS14/105/OHK3/1T/18, No. SGS15/122/OHK3/1T/18, by the Ministry of Education, Youth, and Sports of the Czech Republic under research programs CESNET Large Infrastructure project No. LM2010005, CESNET E-infrastructure project No. LM2015042, by the European Union in the context of the BEBA project (Grant Agreement: 644122) and by the projects TA03010561, TH01010229 funded by the Technology Agency of the Czech Republic.

I would like to express thanks to my colleagues from the Liberouter group, namely Ing. Lukáš Kekely, Ing. Jan Kořenek, Ph.D., Ing. Martin Žádník, Ph.D., Ing. Tomáš Čejka and others, for their valuable comments and motivation in my research.

My greatest thanks go to Petra and my family members for their infinite patience, care, love and support. Finally, I would like to greatly thank to my father MVDr. Zdeněk Benáček for his love, patience and amazing childhood because he suddenly died after the long disease on 3 December 2015 .

Dedication

To Petra and my family for their endless patience and support

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

Miscellaneous Abbreviations

CAM	Content Addressable Memory
CESNET	Czech Educational and Scientific Network
CPU	Central Processing Unit
DFS	Depth-First-Search
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HFE	Header Field Extractor
HLL	High-Level Language
HLS	High-Level Synthesis
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
LAN	Local Area Network
LPM	Longest Prefix Match
LUT	Lookup Table
MAC	Media Access Control
MPLS	Multiprotocol Label Switching
Mbps	Megabit per second
NAT	Network Address Translation
NIC	Network Interface Card
NPU	Network Processing Unit
P4	Programming Protocol-Independent Packet Processors
PGR	Parser Graph Representation
QoS	Quality of Service
RX	Receiver
SDM	Software Defined Monitoring
SDN	Software-Defined Networking

SW	Software
TCAM	Ternary CAM
TCP	Transport Control Protocol
TX	Transmitter
UDP	User Datagram Protocol
VHDL	VHSIC Hardware Description Language
VLAN	Virtual LAN
VXLAN	Virtual Extensible LAN
XML	Extensible Markup Language

Number Sets

$\mathbb{N}$	Natural numbers set
$\mathbb{N}_0$	Natural numbers set $\cup \{0\}$

Common Mathematical Functions and Operators

$\mathbf{b}$	Vector of n elements, $\mathbf{b} = (b_0, b_1, b_2, \dots b_n)$
b_i	the i^{th} element of vector $\mathbf{b}$
$\text{len}(\mathbf{b})$	Number of elements of vector $\mathbf{b}$
$\max(\mathbf{b})$	Maximal element of vector $\mathbf{b}$
$\mathcal{O}(x)$	The big $\mathcal{O}$ notation
$x \ mod \ n$	The integer result of x after division of n
or	Logical *or* operation
and	Logical *and* operation
$log_b(x)$	The logarithm of a number x to a base b
a^b	Power function where a is a variable and b is the exponent (or the power)
$\lceil x \rceil$	Round x upwards. Return the smallest integer value that is not less than x

Chapter 1

Introduction

1.1 Motivation

OpenFlow [3], as the most popular embodiment of Software-Defined Networking (SDN), provides a way to network dataplane configuration at runtime. The OpenFlow specification strictly defines a set of supported protocols and actions for further processing of incoming traffic (i.e., switches are still mostly fixed). However, modern requirements on networking hardware have a dynamic character and administrators of high-end networks want to react to new protocols, security threats, novel approaches in traffic engineering, and so on. This isn't feasible with static network hardware and leads to the need to replace the hardware more often than desired.

The fixed behavior of network devices isn't also suitable for further scaling of data centers because such technological solutions need to react to actual demands on network infrastructure. This need is an impulse for developers and manufactures of networking hardware because they are motivated to provide highly reconfigurable platforms. The example of such solution is not only the OpenFlow switch but also the Facebook's 6-pack [4] or Wedge [5]. Both solutions are equipped with an ASIC chip from Broadcom (for processing of network data at high-speed) and control software which is used as a substrate for implementation of control logic. Such solution can be easily reprogrammed to serve as a router, switch, or network filter.

However, such devices cannot be later extended with support of novel classification, parsing, or packet processing functionality at high-speed (i.e., they are highly reconfigurable but still fixed). Due to this, developers and manufactures of network hardware are motivated to equip network devices with reprogrammable chips like FPGAs. This platform connects the flexibility of software with parallel nature of hardware into one compact package. There are also available hardware platforms [6] for development of network applications, including frameworks for rapid prototyping [7].

The behavior of implemented functionality (or we can say hardware) is traditionally defined by the Hardware Description Language (HDL) which is not easy to learn. Moreover, development and debugging in such language can be time consuming. Due to this, higher

abstractions like C/C++ or Python are being used. However, these abstractions typically suffer from performance issues and it isn't easy to describe high-speed network engine in such language. On the other hand, these approaches are suitable for developers who don't have experience with HDL languages. We feel that programming of FPGAs using the abstract description is very beneficial because user, developer, or network administrator can be focused on network application and doesn't need to care about details of HDL language.

1.2 Goals of the Dissertation Thesis

The goals of my dissertation thesis are summarized in the following list:

1. To propose an architecture of network device which is suitable for automatic generation from an abstract description and is capable to implement high-speed packet processing devices.
2. To identify and provide building blocks which are suitable for automatic generation and are capable to work at high throughput.
3. To introduce a transformation process for generation of a high-speed network device from the abstract description.
4. To evaluate reached results from the view of flexibility, FPGA resources and throughput.

1.3 Structure of the Dissertation Thesis

The dissertation thesis is organized into following chapters:

1. *Introduction* describes the motivation behind our efforts together with our goals. There is also a list of contributions of this dissertation thesis.
2. *Background and Related Work* introduces the reader to the necessary theoretical background and surveys the current state-of-the-art in high-speed computer networks, together with languages for description of packet processing devices.
3. *Architecture of a Network Device* introduces possible targets, details of FPGA platform and it provides the overview of our architecture of network device which is suitable for automatic generation from the abstract description. The chapter also introduces our approach behind the mapping to the proposed architecture.
4. *Parser and Deparser Architecture* provides further details of input/output network blocks (parser and deparser) together with experimental results. This chapter also discusses our transformation process from the abstract description to the architecture of these blocks.

5. *Match+Action Processing Pipeline* describes our architecture of match and action engine which is based on mapping of parsed headers to executed actions. The text also provides details of mapping from the abstract description to the proposed architecture of match and action engine. The section also discusses the usage of current High-Level Synthesis (HLS) tool for description of processing engines at speed of 100 Gbps.

6. *Use Case Study* introduces results for three use cases of generated pipeline, including Match+Action, to demonstrate the flexibility and easy extensibility with new actions and protocols. This chapter provides results of required resources, throughput and possible speedup in development.

7. *Conclusion* summarizes the results of our research, suggests possible topics, and concludes the thesis.

Chapter 2

Background and Related Work

The following section discusses the actual approaches in the area of packet parsing, classification and data processing. These operations are tightly bound and they need to be solved as effectively as possible because they are the keys for the effective and fast data processing in high-speed computer networks. The section also introduces available abstract descriptions of packet processing devices.

2.1 Problem Statement

In this section, we define three fundamental network operation groups, which are the core components of each network device. From the packet processing point of view, we can define three groups of operations:

1. Data Extraction/Packet Parsing
2. Classification
3. Data Processing (we will talk about stateful processing)

The first group, Data Extraction, must precede all other groups like classification, routing, filtering of incoming traffic, and so on. Each incoming packet must pass through some sort of parsing block to understand what is transferred inside the network packet. The examined data is then used in next stages which are related to classification or data processing. Therefore, the parsing is the first action which is performed on incoming network packet and it has to be carefully designed to support the expected packet rate of network interface. For example, the maximal packet rate for 10 Gbps Ethernet with minimal packet length (64 bytes) equals to 1,488,096 packets per second. The network device should be able to process it without any problems. It can be otherwise a bottleneck of the system. A simple parsing example can be extraction of the IP address, TCP ports, or MAC address.

Classification is tightly bound to packet parsing and it is typically the second operation in the packet processing chain. The packet classification means the categorization of incoming packets into classes. The extracted protocol fields form an identifier, which is used for searching of processing rule. Rule is typically stored in a database of rules. This database is searched with packet identifier for the best matching record. Matched rule contains the identifier of the action, which is to be performed on the packet. For example, the network switch extracts the destination MAC address which is used for searching of an output port.

The last stage of the packet processing is the Data Processing block. This block is typically designed to perform some operations like data changing, advanced filtering based on pattern matching, and so on. This stage of packet processing typically accepts read rule from the classification and parsing stage to perform a predefined operation which can be set from a software tool or it can be hard-wired in the networking device. The graphical representation of the packet processing concept is shown in Fig. 2.1. In some cases, we can have more classification and action stages to implement more complex operations like reaction to modified set of header fields (e.g., filtering of traffic behind NAT) and so forth.

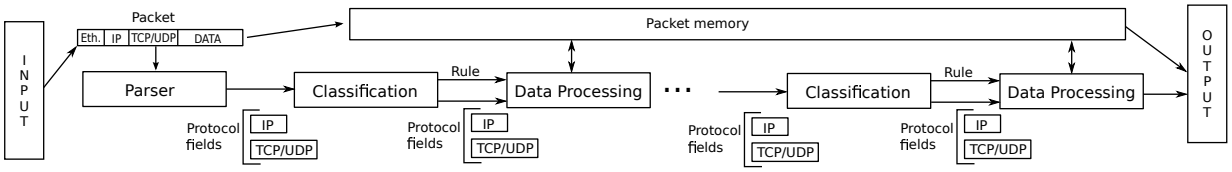


Figure 2.1: Base packet processing pipeline.

2.2 Fundamental Network Operations

The following text describes the current state-of-the-art of briefly introduced groups from the packet processing point of view. We introduce the cutting edge solutions because they are the starting point for our further research.

2.2.1 Parsing

Network communication is divided into layers. Each layer has some task which is required for success delivery of network data. The typical structure of a packet consists of a stack of protocol headers and payloads. Each protocol adds its own protocol header which is understandable by the same network layer on the other device. This is shown in Fig. 2.2. The main task of the packet parsing is the extraction of protocols headers, which are added during transition of the conceptual OSI model. Typical extracted headers are IPv4 [8], IPv6 [9] , TCP and UDP [10].

As we mention in previous section, the Data Extraction is important operation of packet processing because it is typically used at each point of network infrastructure like

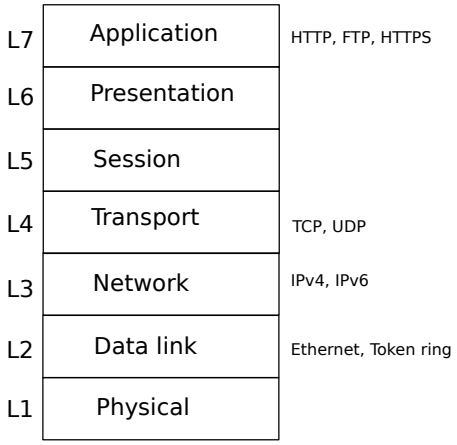


Figure 2.2: Layered OSI Model; It is a conceptual model of network communication. Each layer has some task which is important for successful data delivery between network services. The figure also introduces examples of typical layer protocols.

routers, switches, firewalls, and so on. Moreover, there are requests for more complex and non-trivial packet parsing. It is also the key operation for packet classification because results from this stage are used for searching for the best matching record. The problem is still getting harder because the speed of network lines and parsing requirements are still changing very dramatically. The main reason for this dramatic change is a growing number of communication protocols, which leads to more complicated protocol stack. This can be demonstrated on parsing of TCP protocol [10] in Fig. 2.3. In our example, we introduce three Ethernet frames with different protocol stacks. This protocol arrangement leads to three various starting offsets of TCP protocol which have to be supported by the parser. Moreover, the situation gets more complicated with Virtual Extensible LAN (VXLAN) [11]. The network virtualization technology is used to solve scalability problems in the area of cloud computing. It is similar to classic VLAN technology [12] but it encapsulates the network packets into UDP (instead of the MAC-based Ethernet frames). The extended encapsulation leads to higher requirements on parser. From historical point of view, we can define two groups of parser implementation — software and hardware based parsers.

Software based parsers are common and cheap variant of network parsers. The main component is typically the processor which extracts interesting data from incoming packets. Parser is typically a computer program which implements this functionality. This approach is also used in these days — mainly for lower network speeds because the final solution is cheap and fast enough for this kind of packet processing. The big advantage of this solution is the flexibility of packet parser because we can react to current needs (parser can be easily replaced by new implementation).

The second group, hardware based, is more powerful solution which is able to process much higher throughput. All modern high-speed network devices contains some special kind of hardware accelerated network parser. The disadvantage of this solution is slower development and higher cost. There are implementation platforms like structured ASICs

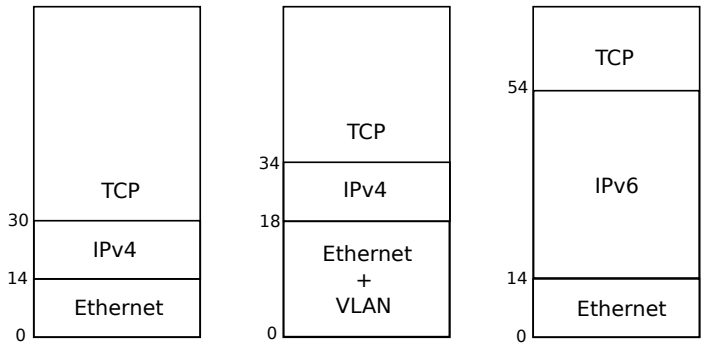


Figure 2.3: The example of packet header arrangement; Number represents the protocol's offset.

or FPGA circuits. The FPGA platform is very popular because of its flexibility, programmability and speed. However, hardware based parser is much harder to develop and maintain than the software version with same functionality. This group of network parsers still faces some challenges to solve. The next section describes the main approaches of hardware based packet parsers.

Braun et al. [13] follow the idea of the layered OSI network model. They introduced a hardware based framework of hand-written wrappers with onion-like structure. For example, the parsing of the IP and UDP protocols consists of parsing of the Ethernet frame, parsing of the IP address and parsing of the UDP protocol. However, the detailed description of the interface is not given in the text. The FPGA implementation is able to achieve the throughput of 2.6 Gbps and it is unclear how the result scales for wider data paths (i.e., higher transfer rates).

Research in this area is very intensive and researches are focused on the usage of the High-Level Synthesis (HLS). The HLS is typically the transformation from a higher level description to a lower level description. For example, Handel-C is very similar to C language. The HLS description is transformed to the HDL (Hardware Description Language) which is typically used for programming of FPGA circuits. There is a general result for the HLS approach which was given by Dedek et al. in [14]. They investigated three different realizations of the same block. There is the comparison of two different realizations in embedded processors (software implementation in customized 16-bit RISC and soft-core $MicroBlaze^{TM}$ processor) and specialized hardware based implementation with usage of Handel-C language. The comparison was made from different points of view: the development time, the estimated frequency and the occupied area. This work demonstrates that usage of processors gives poor results. The average throughput equals to 712 Mbps in the case of custom RISC processor, 83 Mbps for the $MicroBlaze^{TM}$ and 1454 Mbps for the Handel-C implementation. The second important demonstration comes from the usage of the HLS — the throughput and short development time can be obtained by usage of the higher language (Handel-C in Dedek's example).

Kobiersky et al. [15] introduce the packet header analysis and field extraction for multigigabit networks. The architecture for packet header processing is generated from

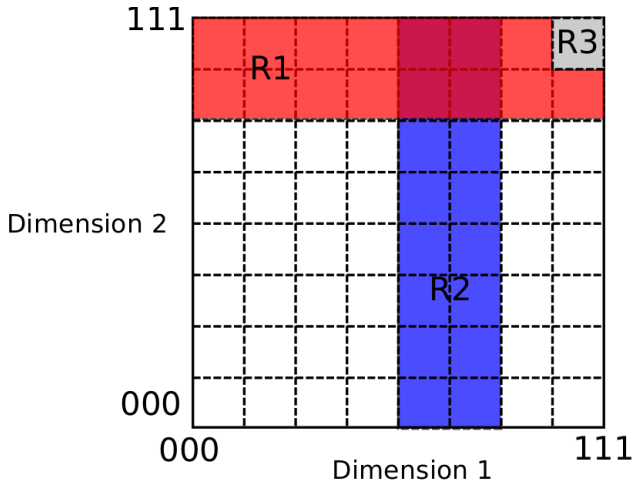
XML protocol scheme. More than 20 Gbps throughput can be achieved using less than 5000 slices of Virtex-5 FPGA. Maximal frequency depends on the internal crossbar which doesn't scale well for wider data paths. However, it is the important result which shows the approach of High-Level Synthesized network parsers in high-speed network area because it allows us to dynamically react to new network protocols. Unfortunately, this approach is harder to use in faster networks (like 40 and 100 Gbps).

A good example of HLS usage was given by Attig and Brebner [16]. They introduce a PP language — a simple high-level language for description of packet parsing algorithms in an implementation-independent manner. The PP language describes the structure of packet headers and methods which define parsing rules. The paper presents the implementation for modern FPGA which is able to accommodate high-speed network processing. Proposed system scales from 1 to 400 Gbps network speeds on Virtex-7 FPGA. The architectural structure of the Brebner's parser is massive microcode controlled pipeline which consumes about 10% of Virtex-7 resources for 1024 bits wide datapath for most common set of supported protocols (Ethernet, VLAN, MPLS, IPv4, IPv6, TCP and UDP). Typical disadvantage of microcode controlled solutions is higher latency compared to pure hardware implementations. In this case, the latency varies from 292 to 548 ns. This is inconvenient in some applications like stock exchange algorithms where the latency is crucial.

Puš et al. [17] proposed a novel architecture of a pipelined packet parser for FPGA based solutions, which offers low latency in addition to high throughput. These two parameters can be tuned to the needs of a particular application. Usage of such solution is very wide. The parser is hand-optimized thanks to the direct implementation in VHDL. The parser consists of analyzing blocks for variety of protocols like IPv4, IPv6, MPLS, and so on. Each block performs the analysis of one protocol — it follows the main idea of layered OSI network model which was discussed in previous text. The communication interface of parsing blocks is generally defined and it can be used for accommodation of new protocols. The *Generic Protocol Parser Interface* (GPPI) consists of the input information which is needed for parsing of single protocol. The GPPI's output interface includes the parsed data and information for the next analyzer block. Unfortunately, the current implementation doesn't contain any HLS support despite the structure of parser is regular which makes it suitable for transformation from general description of parsing process. The parser consumes about 1.19% of Virtex-7 FPGA resources to achieve the throughput over 100 Gbps and 4.88% to achieve the throughput over 400 Gbps for typical set of supported protocols.

2.2.2 Classification

The packet classification can be viewed as a geometrical problem of multi-dimensional discrete space searching. Each dimension represents one parameter which is used for the classification. For example, the most common classification is using five dimensions — source and destination IP addresses, source and destination ports and type of transport protocol. If we combine all these parameters together, we get a five-dimensional search space where each point defines one packet (i.e., one combination). We can assign a rule to each point (or points) of given discrete space. The process of packet classification, from



Rule	Dimension 1	Dimension2
R1	*	11*
R2	10*	*
R3	111	111

Figure 2.4: Geometrical representation of multi-dimensional search space; Asterisk (*) represents any possible bit suffix.

all rules containing (or we can say matching) the packet, selects one rule with the highest priority. The example of geometrical representation of multi-dimensinal classification is shown in Fig. 2.4.

Any searching problem can be solved by searching through all available entries. We can identify two approaches — sequential and parallel. The first group, sequential approach, is performed by comparing the parsed headers with all the entries in the list (memory). The disadvantage of this approach is slow search for large set of rules because the asymptotic complexity of the linear probing is $\mathcal{O}(n)$. We can use a Ternary CAM (TCAM) engine which is used in product devices for parallel multi-dimensional packet classification. TCAM is a specialized memory organized in rows. Each row is composed of number of simple memory cells, where each cell can be set to three different values 0,1 and x (*don't care*). The row matches the lookup key if memory cells with 0 and 1 are equal to the corresponding bit of lookup key. The *don't care* positions are ignored, both values (0 or 1) are allowed. This memory is capable to match all rows in parallel. However, the complexity, cost and power consumption grows with a number of inserted rules. More details about this approach can be found in [18].

The decomposition based methods derive the result of classification by aggregating the individual results from searched domains, where the Cartesian product is typically used during the aggregation phase. These methods offer high throughput but require a big

amount of storage space. The primary challenge is how to efficiently aggregate the results of the single field searches. The required storage space can be reduced by usage of pseudorules which minimize the result of the Cartesian products. It was introduced by Dharmapurikar et al. in [19]. The pseudorule is a generated rule which handles situations for packets which are not directly defined by the rule but it is needed to obtain the correct result of the classification. However, this set of rules and pseudorules can be quite large. For the minimization of generated pseudorules, the Prefix Filtering Classification and Prefix Coloring Classification algorithms were introduced by Puš in [18]. These approaches are able to minimize and eliminate a big amount of pseudorules which leads to effective hardware implementation, capable to accommodate more rules and needed pseudorules. However, the detailed description of pseudorule concept is out of scope of this work.

Most of the classification architectures and approaches are based on direct mapping of this problem to decision trees. *HiCuts* [20] and *HyperCuts* [21] are the main representatives of this approach. They construct a decision tree where inner tree nodes divide the space by hyperplanes into subspaces (e.g., it "cuts" the space into small subspaces which will be searched for the rule). The *HyperCuts* algorithm is faster because it allows cutting on multiple fields per one step. However, the depth of the tree depends on a particular set because the worst case represents the longest path from a root to a leaf. The throughput strongly depends on the passed rule set (i.e., generated tree).

Yuaxuan et al. [22] introduce a tree based multi-dimensional packet classification on FPGA which is capable to process traffic at speed of 100 Gbps. They used the *HyperSplit* [23] algorithm for building of optimized k-d classification tree. The algorithm uses heuristic for selection of most efficient splitting point on a specific protocol field. The paper also discusses the architecture and mapping of *HyperSplit* tree to the FPGA implementation with some optimizations for accommodation of more classification rules. The proposed architecture is able to classify 10K rules at speed of 118 Gbps on Virtex 6 FPGA.

Fong et al. [24] introduce another tree based classification approach which is capable to hit terabit speeds. Authors introduce the *ParaSplit* algorithm which uses a rule set partitioning algorithm. The algorithm is based on range-point conversion to reduce the overall memory requirements. Authors also optimize rule partitioning by applying of Simulated Annealing technique. They implement the architecture for FPGA which allocates separate search engine for each created set. Therefore, the architecture uses the native parallelism for searching of the most suitable rule in more sets in the same time. One proposed engine is able to process 10K rules at speed of 120 Gbps on Virtex 5 FPGA. The terabit classification is possible thanks to instantiation of more search engines on single chip which allows distribution of incoming traffic.

The Grid-of-Tries [25] is the example of hierarchical-trie approach. It is optimized for two dimensions but it can be extended for more dimensions. The algorithm uses an unibit trie [26] for classification of one dimension. The result of the first trie is a pointer to the second trie, where the next dimension is being processed. For example, consider two dimensions — source and destination IP address prefixes. The initial trie is constructed from the source address prefixes. For each source address prefix node is constructed the destination address prefix trie with associated source IP address prefix. Therefore, the

Grid-of-Tries consists of initial source address prefix trie with connected destination IP address tries. Search starts from the source address trie and continues with destination address trie until the desired rule is reached in a leaf.

The most suitable algorithms, capable to hit 100 Gbps, are based on building of decision trees (e.g., *HiCuts*, *HyperCuts* or *HyperSplit*) because these implementations can be effectively mapped into FPGA. Unfortunately, the tree based algorithms have a disadvantage related to duplication of some rules which leads to higher memory consumption. The reason of this behavior is the fact that some rules can have an intersection with two or more rule sets. Therefore, there is a need to duplicate such rules and add them to all related sets. This disadvantage is solved by a different classification approach which is based on Cartesian product of individual results from classification engines. The result of this operation is used for searching in a rule set (i.e., we need to solve if any result of Cartesian product belongs to a known rule). Advanced algorithms of this approach allow to save available memory which has an impact on number of supported rules. However, the Cartesian product based algorithms are not suitable for usage in speeds above 10 Gbps. To our best knowledge, the fastest available solution for this approach was introduced by Kekely M. in [27]. The main idea of author's architecture is based on DFCL [28] algorithm. The proposed results also show that the Cartesian product based algorithms are not suitable for implementation in high-speed network devices capable to hit 100 Gbps.

2.2.3 General Processsing

The data processing phase is typically the last stage of packet processing which is made after parsing and classification. It can be a general operation which is related to change of header field, stateful processing, computation of statistics, advanced filtering based on pattern matching, and so on. Fundamentally, we can identify two groups of general processing: stateless and stateful.

The first group, stateless processing, is typical for situations when the next operation (or packet processing) is not based on previously stored state of the flow. An example of this processing can be insertion of VLAN or MPLS tag based on result from the classification stage, or rewriting of IPv4 address during static NAT.

The second group, stateful processing, is more complex because next packet processing is based on the actual state which is read from internal memory. Therefore, this concept can implement simple or complex Finite State Machine (FSM) behavior for processing of network traffic, where actual state and required data are stored in internal memory (state table). Some stateful processing engines can use the state of connection during classification phase (i.e., we need to connect the state memory before classification engine). In such case, parsed headers are transferred together with read state data to following blocks (*Lookup* and *Update*). The arrangement of state memory depends on needs of implemented application. The main concept is shown in Fig. 2.5. Incoming headers and corresponding state are passed to the *Lookup* block where the engine looks for the most suitable action. After that, the block passes required data (headers, state and selected action) to the *Update* block where the selected action is executed. The action can be related to data change,

update of the state data, and so on. The last step, write back of new state data, is optional and it depends on implemented application.

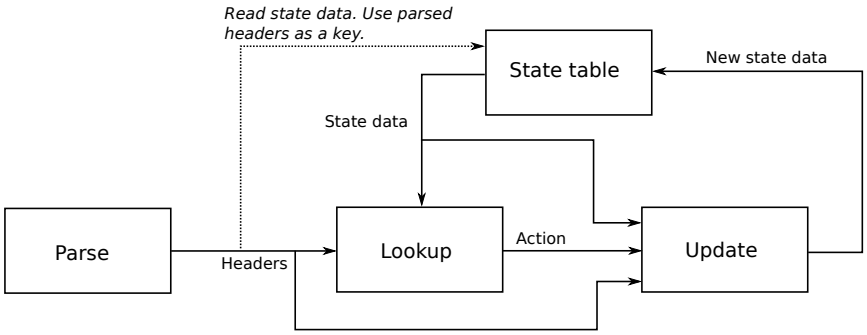


Figure 2.5: General scheme of stateful processing.

Application area of stateful processing is wide and we cannot introduce all possibilities. However, we select a frequent application to provide the basic overview. Traffic measurement (also called as per-flow statistics collection) is a typical example of this approach. It is required in many network application like security, accounting, real-time traffic analysis, and so on. We provide more details about this kind of processing in following text.

Traffic Measurement

Fundamentally, the traffic measurement involves counting number of packets (update operation) which meet some criteria. Moreover, the update criteria can be much more complex. For example, we can use statistical methods for detection of anomalous traffic. The traffic is measured in terms of *flows*. The flow refers to a set of packets which have the same n-tuple in their header field. The n-tuple typically consists of five members: {*sip,dip,spt,dpt,prt*} where, *sip* and *dip* stands for source and destination IP, *spt* and *dpt* stands for source and destination port. Finally, the *prt* stands for the type of L4 transport protocol such as TCP or UDP.

NetFlow is considered to be a traditional measurement scheme. It is based on storage of per-flow records on high-density media. This stored data is processed later to find the answer on the query. This technology requires three components — *NetFlow exporter*, *NetFlow collector* and *NetFlow statistics storage*. *NetFlow statistics* are sampled from network devices such as routers, switches or network probes, to network collector.

All aggregated statistics are stored by the *NetFlow collector* to the *Per-Flow statistics storage*. The most modern collectors and exporters support insertion of information from application layer (L7) which makes user queries much more complex. Discussed architecture is shown in Fig. 2.6. Measured statistics are processed by usage of queries which are created by the operator. The answer of the query is computed on a *NetFlow collector* (for example the number of received bytes of flow with specific IP address and source TCP port). The main disadvantage can be a higher latency of detection.

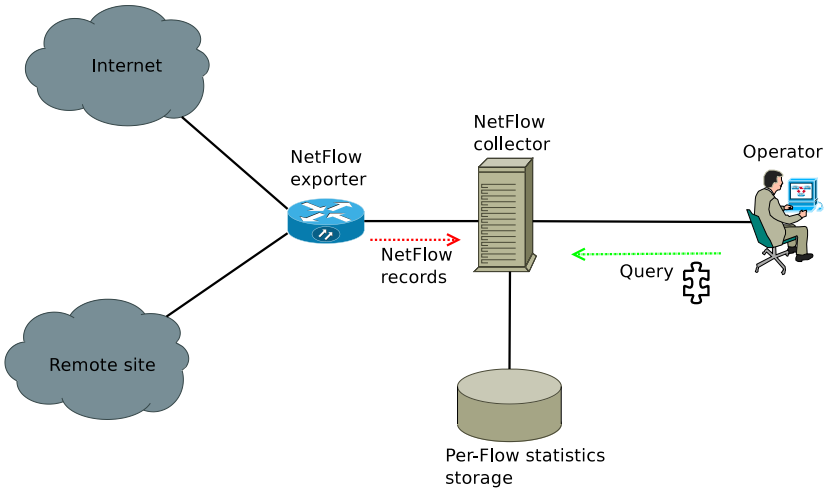


Figure 2.6: Brief NetFlow architecture.

The *Software Defined Monitoring* [29] is the example of advanced flow based monitoring with hardware accelerated extension. The main idea of this approach is to offload the uninteresting flows into the hardware accelerated analyzer while the interesting flows are sent to the software for deeper and more complex analysis. It eliminates the lack of scalability for faster interfaces. Moreover, it connects the world of high-speed networking and HLS. This connection is based on our research which was introduced in [30]. We demonstrated the ability of FPGAs and HLS tools available today to extend the functionality of the hardware by new instructions (new types of aggregation functions). While the typical use of HLS is the synthesis of digital signal processing algorithms, we employ a C/C++ to VHDL synthesizer to implement the instructions of the application specific processor. We implemented the processor infrastructure in VHDL and prepare it for a wide variety of instructions. Finally, we evaluate the approach by implementing five different instructions (two of them being NetFlow like, three non-NetFlow) in High-Level Language (HLL) to assess the versatility of the HLS for the task of processor instruction description. The SDM consists of two main parts (firmware and software) connected together via PCI Express

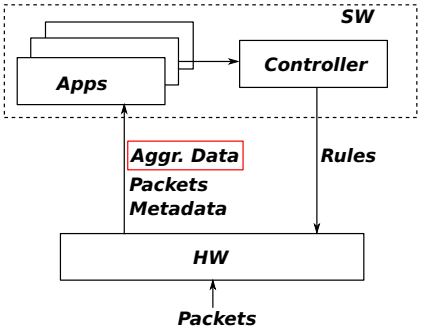


Figure 2.7: Brief SDM Architecture.

bus. Both parts are tightly coupled together to allow a precise control of the hardware processing on the per-flow basis. The software part of SDM consists of the controller and monitoring applications. Advanced monitoring tasks, such as analysis of application protocols, are performed in the monitoring applications. The management of the hardware (removing and insertion of the processing rules) is performed in the controller. The SDM Processor hardware is controlled by instructions stored in rules. The instruction tells the hardware what action must be performed for each input packet. The hardware passes the data to the software in the form of packet metadata (extracted information from the packet header) or aggregated flow records (NetFlow for example). Whole received packet can be also sent to the software for deeper analysis. Graphical representation of the SDM concept is shown in Fig. 2.7.

2.3 Languages and Abstractions of Network Devices

The following text introduces modern languages and abstractions of packet processing devices, together with code examples.

2.3.1 OpenFlow

OpenFlow [3], as the most popular embodiment of the ideas of Software-Defined Networking [31], provides a way to fill the lookup tables of switches at runtime. The OpenFlow specification defines exactly which protocols are supported by switches. At the same time, it is not possible to change the set of supported protocols or actions — switches are fixed (or at least they appear so to the OpenFlow controller).

Architecture of packet processing device with OpenFlow support is given in Fig. 2.8. Whole concept contains several components:

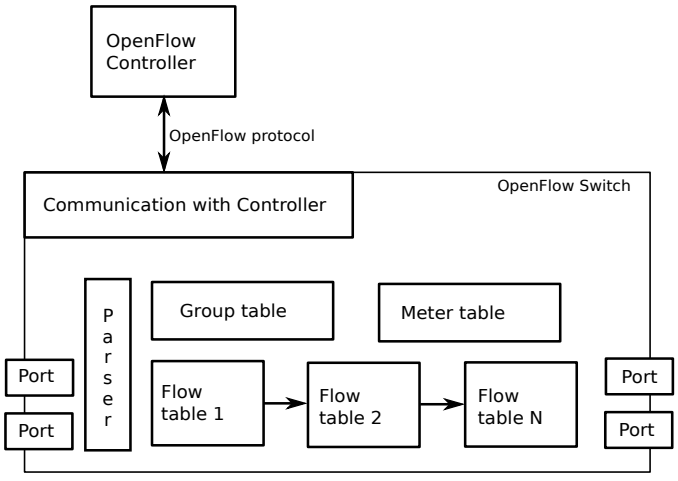


Figure 2.8: Architecture of packet processing device with OpenFlow support.

- **OpenFlow Controller** is a centralized component which controls more OpenFlow switches. It is used for configuration of OpenFlow switches at runtime. That is, it defines mapping between protocol fields and used actions. The action can be related to protocol field change, packet drop, forwarding of processed packet to next table, and so on. Therefore, the OpenFlow Controller defines a behavior of forwarding plane by filling the tables with rules. The example of OpenFlow Controller is Ryu [32] which is available under open source license.

- **OpenFlow communication protocol** gives the access to the forwarding plane of SDN switch. It also defines a set of supported protocols, actions and tables. These sets typically depend on version of OpenFlow. The latest specification is from April 2015 (OpenFlow version 1.5.1 [33]).

- **OpenFlow Switch** is the realization of forwarding plane (software or hardware based implementation). All capabilities are defined by supported version of OpenFlow protocol. OpenFlow Switch typically consists of several tables. As we noticed in previous text, the behavior of the forwarding plane is defined by rules which are inserted to lookup tables at runtime. The most important is the *Flow table* which is used for implementation of match and action functionality. That is, it uses protocol fields of processed packet for identification of the most suitable action which is executed in the same table. OpenFlow switch typically contains more *Flow Tables* which are used for implementation of more complex match and action behavior. The *Group table* supports sets of actions for flooding as well as more complex forwarding semantics (e.g., multipath, fast reroute, and link aggregation). Finally, the *Meter table* consists of per-flow meter entries, which enable OpenFlow to implement various QoS operations.

The packet, entering OpenFlow switch, is processed by the parser which identifies available protocol fields. These fields are used as a key for searching of most suitable action in first *Flow table*. The searched action is executed if the *Flow table* founds matching record. In the case of tables miss, the *miss-action* handler is activated. Typically, the non-matching packet is sent to SDN Controller for further processing which can lead to insertion of new rule (i.e., the logic of network application is implemented in controller and it decides about the future of given flow).

The detailed description of SDN and OpenFlow is out of scope of this work. However, we provide basic pros and cons of this approach. The main advantage is the direct programmability of forwarding plane from *OpenFlow Controller*. Simply said, the controller defines the logic of network forwarding. This logic (or some part of it) is translated to the form of rules which are uploaded to tables in *SDN Switch*. The second advantage is connected with central management of the SDN network which is beneficial for maintaining of global view on the network. As we noticed before, the main disadvantage is connected with limited set of supported protocols and actions which cannot be extended (i.e., any new functionality has to be emulated in software). This behavior is inconvenient in high-speed networks because network device has to process big amount of packets in a short

time. This need leads to an extensible forwarding plane which bypasses the emulation of new functionality in software.

2.3.2 Gorilla

Lavasani, Denninson and Chiou introduced the Gorilla methodology in [34]. This methodology is used for compilation of high throughput network processors from abstract description. The main goal is to abstract network engineers from underlying hardware. The Gorilla methodology defines canonical and general architecture for implementation of variety of network applications. The Gorilla's compiler accepts an abstract description which is translated to a Verilog source code. The translation process uses highly parameterized templates which are created by a skilled HDL designer. All modules can be connected in pipelined or multithreaded manner. The multithreaded hardware supports more "threads", where each thread does a specific unit of work.

The proposed approach allows to find a trade-off between consumed resources and reached throughput with retained generality. Network engineer can describe the packet processing in a language similar to C notation. The following example is taken from [34]:

```
IPv4_check() {
    status = IPv4_header_integrity_check(Header);
    if (status == CHKSUM_OK)
        Next_step = IPv4_lookup;
    else
        Next_step = Exception;
}

IPv4_lookup() {
    Da_class = lookupx.search(Header.IPv4_dstaddr);
    Sa_class = lookupy.search(Header.IPv4_srcaddr);
    if (Da_class == NOT_FOUND)
        Next_step = Exception;
    else if(Sa_class == NOT_FOUND)
        Next_step = Exception;
    else
        Next_step = IPv4_modify;
}

IPv4_modify() {
    if((IP_update_fields(Header) == ZERO_TTL))
        Next_step = Exception;
    else {
        Dport = Da_class.dport;
    Next_step = Emit;
    }
}
```

Authors of the proposed methodology provide performance results for the FPGA implementation. The Xilinx Virtex-5 TX240T FPGA (used on NetFPGA-10G board) was used for implementation of the IPv4 lookup. The Xilinx Virtex-6 HX380T FPGA was used for implementation of 50 Gbps multi-protocol stack (IPv4, IPv6 and MPLS). Finally, the Xilinx Virtex-7 VHX870T was used for implementation of 100 Gbps multi-protocol engine. All cores were able to hit the target throughput on 64 byte Ethernet packets at frequency of 100 MHz. These results make the Gorilla very suitable for implementation of high-speed network processors.

However, the proposed methodology uses templates of processing engines and packet parsers which makes the Gorilla somewhat static. The example of this static behavior is the inability to extend the set of supported protocols and packet parsers during development of network application (i.e., Gorilla supports the limited set of protocols which are supported by the HDL template library). The Gorilla methodology is intended to be used in FPGA only.

2.3.3 SDNet

The SDNet [35] is a commercial solution from Xilinx. The PX language is used for abstract description of packet processing devices. We provide the example of Ethernet parsing that has been taken from [35]:

```
class OF_parser::ParsingEngine (9000,4) {
    // Tuple class for extracted data
    class Flow :: Tuple(inout) {
        struct flow_s { // OpenFlow 12-tuple
        port:3,
        dmac:48,smac:48,type:16, // Eth
        vid:12,pcp:3, // VLAN
        sa:32,da:32,proto:8,tos:6, // IP
        sp:16,dp:16 // TCP
    }
}

Flow fields;
// Section class for Ethernet header
class ETH :: Section {
    struct {dmac:48, smac:48, type:16}
    method update = {
    fields.dmac = dmac,
    fields.smac = smac,
    fields.type = type
    }
method move_to_section =
    if (type == 0x8100) VLAN
```

```
    else if (type == 0x0800) IP
    else done(1);
    method increment_offset = size();
}

// VLAN, IP, TCP classes follow ...
// Similar style to ETH class
}
```

The complete parser declaration contains a declaration of the header format, together with methods that define selection of next processed protocol (i.e., it defines the type of next used protocol) and computation of the next starting offset. The rule searching engines supports three classes of algorithms: Content addressable memory (CAM), Longest-prefix match and ternary CAM. Users of SDNet can define a composed design from parsers and user defined processing engines. Such definition requires specification of an interconnection pattern between used modules. The following example of MPLS LSR system with Secret Sauce user engine has been taken from [35] (interconnection is defined in *connect* method):

```
class MPLS_LSR::System {
    // Input and output interfaces
    Packet_input instream;
    Packet_output outstream;

    // Sub-engines
    MPLS_Classifier classifier;
    Secret_Sauce relay;
    MPLS_Editor editor;

    // Interconnections
    method connect = {
    classifier.packetin = instream,
    relay.in = classifier.packetout,
    editor.packetin = relay.out,
    editor.fields = classifier.fields,
    editor.op = classifier.op,
    outstream = editor.packetout
    }
}

// ...
class Secret_Sauce :: UserEngine {
    Packet_input in;
    Packet_output out;
}
```

The SDNet proposes a flexible solution capable to scale line rates from 1 Gbps to 100 Gbps. The PX language is a modern and powerful concept for description of high-speed packet processing engines. Xilinx also introduced a translator from P4 (se later) to PX language in [36]. However, the Xilinx SDNet is closed system which doesn't allow implementation of novel hardware approaches (i.e., all changes have to be made by Xilinx). This is complicated for network researchers because they have to wait for implementation of novel algorithms and hardware architectures in SDNet. Therefore, there is a need for open and customizable solution which is suitable for translation from high-level description to HDL language.

2.3.4 OpenState

OpenState [1, 37, 38] is a research effort focused on development of a stateful data plane and API for SDN. Authors of this approach proposed an extension to current OpenFlow abstraction that uses state machines implemented inside switches to reduce the need to rely on remote controllers. It also introduces usage of *eXtended Finite State Machines* (XFSM; more in [39, 40]). This extension allows implementation of complex stateful behavior like port knocking, DDoS detection/mitigation, and so on.

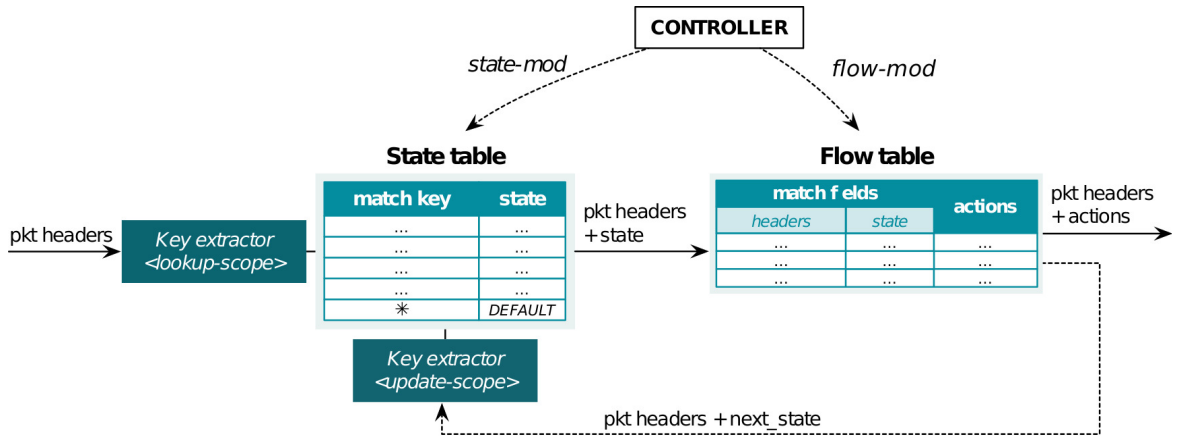


Figure 2.9: OpenState concept (taken from [1]).

Basic OpenState concept is shown in Fig. 2.9. As defined by OpenFlow, a packet is processing through the pipeline of tables where each table provides the match and action functionality. Firstly, the entering packet is processed by *Key extractor* which produces a key based on configuration of *lookup-scope*. The key is used for identification of state in *State table*. The matched state data is transferred together with packet headers to *Flow table*. This table is extended with matching of state which can be understood as a virtual protocol field (i.e., it isn't the part of a protocol but it is used for matching). Outputs from *Flow table* are following — (1) packet headers with suitable action and (2) updated state information. The update is written to the *State table* based on configuration of *update-scope*. The pros and cons are summarized in following lists:

- **Pros:**
  - Based on existing proven standard (OpenFlow).
  - Powerful stateful processing.
- **Cons:**
  - Feedback loop complicates implementation. It may require synchronization among CPU threads, pipeline stalls in FPGA/ASIC.
  - Non-extensible protocol and action support.

2.3.5 P4

P4 (Programming Protocol-independent Packet Processors) [41, 42] is an open source, high-level and platform-agnostic language. It represents a recent contribution to the broader idea of Software-Defined Networking (SDN) and its ecosystem. Its main purpose is to provide a way to define packet processing functionality of network devices, paying attention to reconfigurability in the field, protocol independence and target platform independence. Using relatively simple syntax, P4 allows to define five aspects of packet processing:

- **Header Formats** describe protocol headers recognized by the device.
- **Packet Parser** describes the (conceptual) state machine used to traverse packet headers from start to end, extracting field values as it goes.
- **Table Specification** defines how the extracted header fields are matched in possibly multiple lookup tables (e.g., exact match, prefix match and range search).
- **Action Specification** defines compound actions that may be executed for packets.
- **Control Program** puts all the above together, defining the control flow mainly among the tables.

All proposed aspects can be mapped to three operations of packet processing. The Header Formats and Packer Parser describes the structure of incoming data and relations between protocols. We can map these two aspects to the Packet Parsing. The Table Specification generally describes the process of classification. In general, the classification uses the extracted data for searching of most suitable processing rule. The Action Specification is used for definition of general processing action. The Control Program specifies the sequence of classification and action stages. Therefore, we can divide the classification and action blocks to more stages which allows us to implement more complex behavior. We also demonstrate the usability and flexibility of this language, compared to existing SDN ecosystem, in [A.13, A.14]. The following text provides examples of syntax and further details of P4 language.

Header Definition

Header Definition is very similar to declaration of structure in C language. The P4 language uses a simple syntax in the form of *name* : *width*. Header Definition for the Ethernet header looks like this:

```
header ethernet {
   fields {
      dstAddr   : 48; // width in bits
      srcAddr   : 48;
      ethertype : 16;
   }
}
```

The description simply lists fields of the packet header and their width in bits. The example above shows a protocol with static header structure, where the header length is the sum of lengths of all fields. This can't be done for protocols with variable header length. The P4 language solves this situation by the header length definition in the form of an expression which uses fields from the protocol header declaration to compute the header length. Header Definition with variable length may looks like this:

```
header_type ipv6_ext_t {
    fields {
        nextHdr     : 8;
        totalLen    : 8;
        frag        : 12;
        padding     : 3;
        fragLast    : 1;
    }

    length     : (totalLen + 1) * 8;
    max_length : 1024; // Bytes
}
```

Parser Definition

Parser Definition is used for the description of relations between protocol headers That is, it defines one state of FSM which describes the parsing process in terms of transitions between protocols. Parser Definition may looks like this:

```
header ethernet eth;
parser parse_ethernet {
   extract(eth);
   switch(eth.ethertype) {
      case 0x8100: vlan;
      case 0x9100: vlan;
```

```
        case 0x0800: ipv4;
        case 0xA100 mask 0xF100 : myProto;
        default : ingress;
    }
}
```

The provided example uses *switch* and *extract* statements. The *extract* statement instructs the parser to examine input packets and to look for data defined in the header. Parsed data is then used in the *switch* statement to determine the next state (protocol) to process. There are also situations when we don't want to use the whole value from the protocol field. The P4 language solves this with the *mask* statement which is used in the *case* statement together with mask value. In our example, the *mask* statement instructs the P4 parser to take the *ethertype* field, perform logical *and* operation between the value and mask. Finally, the result is compared to *0xA100* value. The *switch* statement can contain not only references to next parsers but it can also contain references to Control Flows. Such references instruct the P4 program to stop parsing and continue with processing of extracted data. The entrance point of each P4 program is the parser with *start* name, for example:

```
 parser start {
    return parse_ethernet;
}
```

Match+Action Table Definition

The Match+Action Table is used for mapping of extracted header fields to most suitable action. The Match+Action Table consists of:

1. **Extracted header fields** - these fields are used for identification of record in Match+Action Table. Each referenced field contains the declaration of algorithm which is used for matching. The list of supported matches is following: *exact*, the *Longest Prefix Match* (LPM), *range*, *valid* and *ternary* match. The *exact* match is working with value without any masking. The LPM is the most common matching algorithm in IP protocol which is based on looking for the longest binary prefix in passed data. The *ternary* match is more general than LPM because it allows us to work with three bit values - logical *1*, logical *0* and *don't care*. Therefore, LPM is a special case of *ternary* match. The *range* match is used for matching on range of values. Finally, the *valid* match checks the presence of used protocol header in processed packet.

2. **List of supported actions** - this declaration contains the list of supported actions which can be used from the particular table.

The example of Match+Action Table definition in P4 language can be following:

```
table filter {
    reads {
        ipv4.srcAddr    : lpm;
        tcp.dstPort     : exact;
    }

    actions {
        PushVlan;
        Permit;
        NoOp;
    }
}
```

The proposed example uses two protocols for matching (all used fields are listed in the *reads* statement). The fist one is the source address of IPv4 protocol which uses the LPM algorithm for matching. The second one, TCP port, uses the *exact* match. Finally, the list of supported actions is defined in the *actions* statement. Actions can be filled with some parameters. These parameters are stored within the record in the Match+Action Table. More details about declaration of actions are situated in following text.

Action Definition

The P4 language defines the list of primitives actions (see the P4 language specification in [2]). These primitive actions can be combined to more complex user actions. Each user defined action can call other user defined or primitive actions. The following example is taken from [2]:

```
action add_mTag(up1, up2, down1, down2, egr_spec) {
    add_header(mTag);
    // Copy VLAN ethertype to mTag
    copy_field(mTag.ethertype, vlan.ethertype);
    // Set the VLAN's ethertype to signal mTag
    copy_field(vlan.ethertype,0xaaaa);
    set_field(mTag.up1, up1);
    set_field(mTag.up2, up2);
    set_field(mTag.down1, down1);
    set_field(mTag.down2, down2);
    // Set the destination egress port as well
    set_field(metadata.egress_spec, egr_spec);
}
```

All action are performed in sequential manner. The action can be imagined as a function in C language (the syntax is similar). The action can accept some parameters. As was mentioned in previous text, these parameters are stored within record in the Match+Action Table and we can configure these parameters during runtime (i.e., configuration of the P4 device from software tool). In our example, the mTag header (user defined) is inserted after

the VLAN header. To do that, we need to copy the ethertype value from the VLAN header to mTag, and setup the VLAN ethertype to 0xaaaa (we want to signalize the presence of the mTag protocol). Finally, we setup the rest of the mTag fields and value of output port.

Control Flow Definition

We defined headers, parser, tables and user defined actions. Finally, we need to define the ordering of Match+Action Tables in program. This behavior is defined in the P4's Control Flow. We start the description with simple example (the code is taken from [2]):

```
control ingress {
    // Verify the mTag state and port
    apply(source_check);
    // If no error from source_check, continue
    if(!defined(metadata.ingress_error)) {
        // Attempt to switch to end host
        apply(local_switching);
        if(!defined(metadata.egress_spec)) {
            // Not a known local host
            apply(mTag_table);
        }
        // Check for unknown egress state or bad retagging with mTag
        apply(egress_check);
    }
}
```

Match+Action Tables are executed using the *apply* statement. The control block typically consists of *apply* and *if-else* statements. Used tables can be selected regarding to hit/miss in previous table, selected action, and so on. Therefore, we can implement a quite complex behavior of packet processing. The full description of all available functionality can be found in [2].

2.4 Summary

We introduced and described three network operation groups — parsing, classification and general processing. The latest and most modern solutions for each operation were described. Finally, we introduced high-level solutions for description of packet processing functionality — Gorilla, SDNet and P4. The Gorilla is one of the first approaches for description of high-speed network devices but the solution is somewhat static in term of extensibility (new protocols and actions). The SDNet is the latest and most modern approach capable to scale from 1 Gbps to 400 Gbps. However, this technology is closed which makes it quite hard to extend it with newest FPGA based hardware solutions from network area. The P4 language is a complex language for specification of modern packet processing devices which is developed under open-source license. The specification also

introduced the P4 language as a platform independent solution which makes it usable not only in FPGAs. We also demonstrate the usability, platform independence and flexibility of this language, compared to existing SDN ecosystem, in [A.13, A.14]. Therefore, we will focus on P4 in following text.

Chapter 3

Architecture of a Network Device

In this chapter, we introduce possible target architectures for mapping of P4 program. Namely, we introduce details about state-of-the-art software implementations capable to process 10 Gbps, Network Processing Units (NPUs) capable to process traffic at speed of 40 Gbps and beyond. The text also provides details about Field Programmable Gate Array (FPGA) which is a common platform for implementation of high-speed network hardware. Finally, this chapter proposes our architecture of high-speed processing pipeline which is suitable for mapping from the abstract description.

3.1 Target Architectures

This section introduces the most common target platforms for implementation of network applications. We introduce software based targets in the beginning of the text. Then, we continue with introduction of hardware based targets which are suitable for processing of high-speed network traffic.

CPU

The most common target is a software tool for ordinary Central Processing Units (CPUs). Such solutions are commonly used because they are flexible and easy to deploy with minimal cost. Moreover, computers can be connected to a large and powerful cluster which is capable to process high-speed network streams. The disadvantage of this approach is poor scalability for processing of network data in real time.

Pedro et al. [43] introduce the solution for analysis of 10 Gbps traffic on commodity hardware with slightly modified network drivers. The target architecture is able to fully classify incoming traffic at speed of 14.2 milion packet per second (Mpps). However, the paper doesn't present any possibility for generation of processing program from abstract description.

Marian et al. [44] introduce the NetSlice operating system abstraction which tightly couples the hardware and software packet processing resources. The NetSlice performs

domain specific, spatial partitioning of system resources like CPU cores, memory and NIC. It also provides a special low latency channel between NIC and user-space application. Using this approach, the user-space application is capable to receive and process network traffic at speed of 10 Gbps.

CLICK [45] is the domain specific language and set of modules for implementation of network applications. The language defines a unified interface for easy connection of all modules which are organized in directed dataflow graph. Modules are capable to construct and modify protocol fields of processed packets. This solution can be easily extended to support new protocol sets. However, this tool is implemented in software which doesn't make it suitable for using in high-performance networks.

The P4 language consortium introduces the P4-HLIR library [46] which is the front end of the P4 compiler, creating Python object model of the P4 program. It is useful for other projects because one can easily continue with implementation of compiler's back end from this object representation. The P4C-BEHAVIORAL compiler [47] is the example of such implementation. It implements the back end for translation from P4 language to C++.

NPU

Another common architecture is the Network Processing Unit (NPU). It is an integrated circuit which has a specific feature set for network application domain. The example of this feature set can be fast and effective implementation of classification engines, pattern matching, data bitfield manipulation, and so on. The NPUs are typically software programmable and parallel devices with specialization on computer networks and telecommunications. The architecture of network processors can be described in many ways. An extended general framework for classifying network processors was suggested in [48] including these five dimensions: (1) Parallel processing approach, (2) Hardware assistance (coprocessors for variety of optimized tasks), (3) Network processor interconnection mechanisms (e.g., on-chip communications), (4) Peripherals, (5) Flynn typology of multiprocessing (SIMD, SISD, MIMD, MISD). The disadvantage of this approach is the limited set of instructions/actions which cannot be extended during run time (i.e., we have to emulate them in software). However, the NPUs are capable to process network traffic at speed of 40 Gbps [49] and beyond in real time.

FPGA

The Field-Programmable Gate Array (FPGA) is a common target platform in these days. It is an integrated circuit which is used for rapid prototyping of hardware. The structure (and thus behavior) of implemented hardware is defined using the Hardware Description Languages (HDL). This platform connects the performance of NPUs and flexibility of software solutions to one compact package (the FPGA is also called as a *programmable hardware*). There are also available FPGA based NICs which are capable to process speeds from 10 Gbps to 100 Gbps (see [6, 50] for further description).

3.1.1 Detailed Description of FPGA

FPGA is a semiconductor device whose structure can be reconfigured. The configuration is specified using the HDL which is translated to bitstream (i.e., the configuration stream of FPGA). Simplified architecture of FPGA is shown in Fig. 3.1.

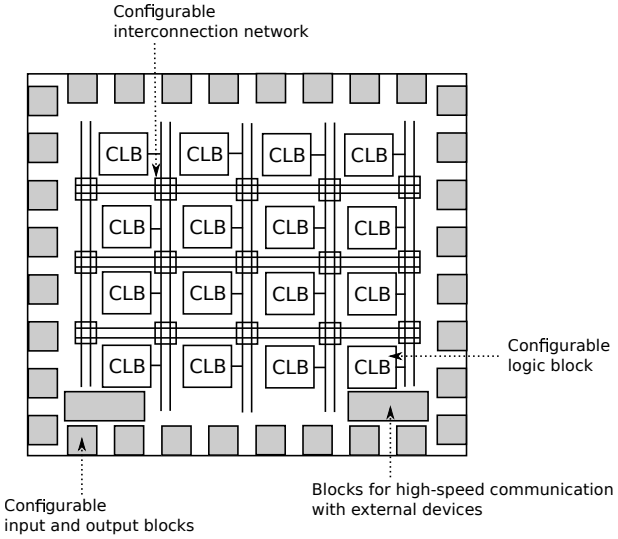


Figure 3.1: Simplified architecture of FPGA.

The FPGA contains Configurable Logic Blocks (CLBs). These blocks are used as generators of logic functions and sequential logic. The generator is realized by a memory which is configured with the truth table of implemented function — Lookup Table (LUT) in terms of FPGA. The CLB also contains the gated D latch (i.e., one bit register) for implementation of sequential logic. Required behavior can be internally selected by multiplexer (e.g., we can select the register and enable the sequential behavior). The brief scheme of CLB is shown in Fig. 3.2. However, each vendor implements different features to CLB blocks.

The number of LUT inputs defines the row count of realized truth table (2^n possible results for n inputs). The most common number of LUT inputs is equal to four or six. Function with more variables is realized by connection of LUTs to a chain. FPGAs are often equipped with more specialized blocks like memories [51], Digital Signal Processing (DSP) blocks [52], processors (like ARM) [53], and so on. These embedded blocks have an influence on final frequency and used resources. For example, we can save CLB logic for implementation of memory or processor. Each mentioned block communicate via configurable interconnection network.

The communication with external devices is performed via configurable input and output blocks. These blocks enable adaptation to external signals and transfer them inside the FPGA. The high-speed communication can be performed via interfaces like RocketIO [54] (line rates up to 3.125 Gbps), GTX/GTH [55] (line rates up 13 Gbps), and so on. These high-speed communication interfaces can be used for attaching of Ethernet transceivers.

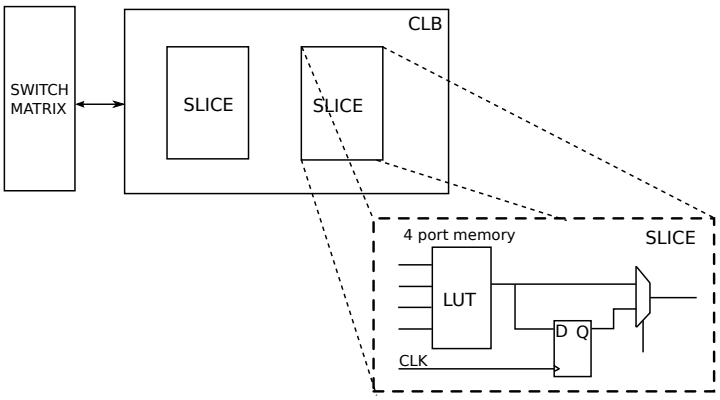


Figure 3.2: Simplified architecture of CLB.

Programmability of FPGAs make them suitable for the research in the area of computer networks because we can use an HDL language and describe a complex behavior of hardware device. It is also used for fast prototyping of ASICs because developers can run and validate the chip's design before production. This platform is also used by other researchers in this area for hardware acceleration of classification, packet parsing, and so on. From provided description of FPGA, we feel that this target is very flexible and we can use it as a target platform for implementation of a generic network device which is being generated from P4 source code. The high-speed networking world offers many challenges regarding architectures and algorithms for implementation of network devices, capable to process traffic at speed of 100 Gbps and beyond. Moreover, the FPGA platform allows to accommodate different applications without any change of physical hardware. Therefore, we want to use the P4 language together with FPGA target for our research.

3.2 Generic Architecture of a Network Device

As we notice before, we chose the FPGA as the main implementation platform for generated devices from P4 description. This platform connects the flexibility of software and parallel nature of hardware together (i.e., the functionality of the hardware can be reprogrammed). The initial idea of P4 device model was introduced in [41, 2, 56]. The text provides more details about this model in Sec. 3.2.1. Unfortunately, the P4 device model isn't suitable for some use cases because the match and action part is strictly defined. We propose the Parser-Deparser model which freely defines a functionality between parser and deparser blocks. The text provides more details about this approach in Sec. 3.2.2. Finally, Sec. 3.2.3 proposes the architecture of generic processing pipeline which is suitable for accommodation of P4 hardware.

3.2.1 P4 Device Model

The P4 device model contains parser, deparser, Match+Action tables and queuing mechanism (see Fig. 3.3). The first module, parser, is used to break the incoming network data into individual header fields. These data fields are passed to the ingress Match+Action tables where packet modification and egress selection is performed. After that, the ingress Match+Action pipeline passes data to queuing mechanism which implements traffic distribution, based on configuration from ingress Match+Action tables. Egress Match+Action pipeline is used for final modification of protocol fields before deparsing, which is performed in the last module. This module is used for construction of network packet back from the form of protocol headers. The proposed model is general enough for wide range of applications. However, it isn't able to implement another advanced architectures for processing of network traffic (like pattern matching or extended stateful processing).

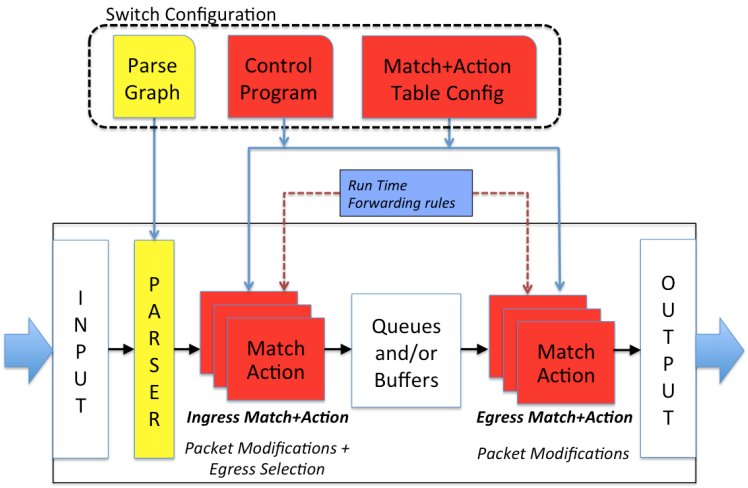


Figure 3.3: Model of P4 device (taken from [2]).

3.2.2 Parser-Deparser Model

Due to the disadvantage described above, we propose more general model which is based on free specification of processing part (simplified architecture is shown in Fig. 3.4). Our model of network device is based on standard P4 model and it consists of three modules. The first module, parser, is used to break the incoming network data into individual header fields (i.e., it has the same functionality like parser in P4 model). The output of this module is a set of structured extracted fields and valid bits. The valid bit is used for presence indication of extracted protocol fields in actually processed packet. The structure of protocol fields

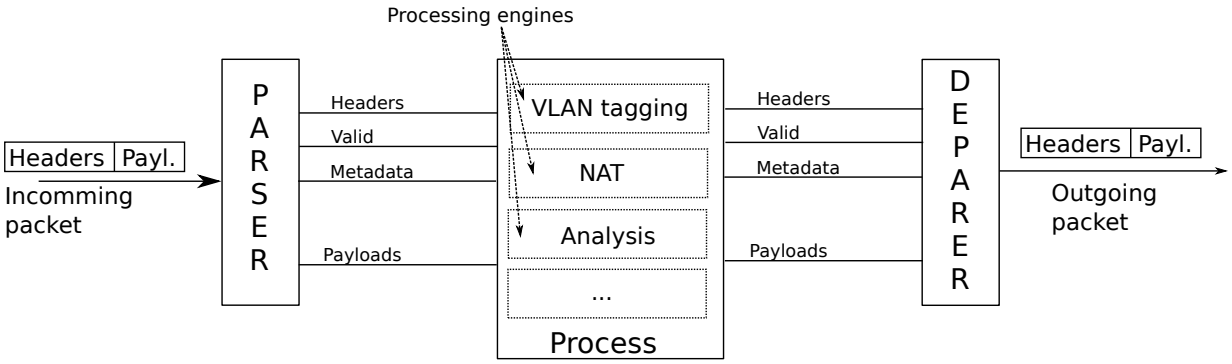


Figure 3.4: Parser-Deparser model.

is based on the P4's Header Format definition. All extracted protocol fields and valid bits are passed to the second module, process, which implements the general processing engine (e.g., VLAN tagging, Network Address Translation, Traffic analysis). This block implements the functionality of accelerated engine. It also sets a validity information for each inserted/not inserted protocol and filters out all unused header data (i.e., this data will not be used during packet assembling). Finally, the last module, deparser, is used for construction of network packet back from protocol headers and valid bits. This approach is general enough for variety of network applications. The Parser-Deparser model is capable to accommodate the P4 device model because the Process part allows mapping of Ingress Match+Action, Queues and/or Buffers and Egress Match+Action to predefined general architecture. This work introduces possible mapping in Sec. 3.2.3.

The VLAN tagging is a classic and common use case in many network applications. Tagging process consists of two main steps — insertion of VLAN header to packet and setting of appropriate header field value of current Ehernet protocol. In the case of Parser-Deparser model, the supported protocol stack is directly represented in the hardware device where each protocol occupies one position on header and valid bus. Therefore, the VLAN tagging operation consists of setting of the VLAN header on header bus, setting of appropriate valid bit (i.e., header will be inserted) and setting of appropriate Ethernet header field value. Finally, the packet is assembled in the deparser block.

The network traffic analysis is another common use case. The analysis is based on investigation of data from extracted headers which are used as an input to analysis function. This function can be a general process like pattern matching, detection of malicious traffic, and so on. The Parser-Deparser model allows easy access to all extracted data in parallel because data are preprocessed in parser stage and put onto header data lines. The traffic analysis algorithm is implemented in process module. This use case isn't easy to implement in standard P4 device model because the structure and set of supported actions is predefined in P4 standard.

In this section, we proposed our Parser-Deparser model. The model can be partially generated from P4 description. More precisely, we can generate the parser and deparser modules from the Header Format and Packet Parser definition, because both modules have

homogeneous and general structure. Our Process block can be generated from P4, but is generic beyond what P4 describe. In this work, we show the ability of our model. We use a P4 language as an input of transformation process which effectively maps the P4 source code to hardware representation. The Parser-Deparser model is published in [A.10].

3.2.3 Mapping P4 to Parser-Deparser Model

The following text brings details of the architecture of processing pipeline in Fig. 3.6 (published in [A.7]). The idea uses our Parser-Deparser model which was introduced in Sec. 3.2.2. The *parser* is used to break the incoming data to individual protocol header fields and payloads. Our architecture also allows synchronization of outside metadata with parsed headers. The outside metadata are generated by input network buffer during receive of packet from network. This structure typically contains input port number and receive timestamp. The parsed headers and outside metadata are passed to the *metadata generator*. This block cooperates with parser on initialization of default metadata values. The generated information is later used for advanced processing of parsed data in our pipeline (i.e., we can use metadata for controlling of functionality of processing elements). Another output from *parser* and *metadata generator* is also the address of first active Match+Action processing element in *Match+Action group*.

All outputs (parsed headers, generated metadata, address of first *Match+Action group* and parsed payloads) from *parser* and *metadata generator* are passed to *check* module. This module performs validation of the checksum field of processed packet and setting of initial values of control path. Some protocols need to be checked for validity of header or payload data because there may be errors during transfer through the network (i.e., we need to detect this situation in processed protocol and pass this information to the pipeline). Our architecture supports processing of protocol with such specification. The example of protocol with checksum field is the Transmission Control Protocol (TCP) which is defined in [57]. Initial values of control path signals are generated from result of the checksum validation, start table address, incoming protocol headers and metadata values. This path is used for controlling of all elements in the high-speed processing pipeline. Typical purpose of this path is propagation of errors, packet discarding, controlling of Match+Action element address, and so on. Finally, all generated control path values, parsed headers and metadata are passed to the *Match+Action group* pipeline (see later). Parsed payloads are stored in large Payload Buffers where each buffer stores payload for one protocol. For example, we can receive a packet which is constructed from IPv6 header,

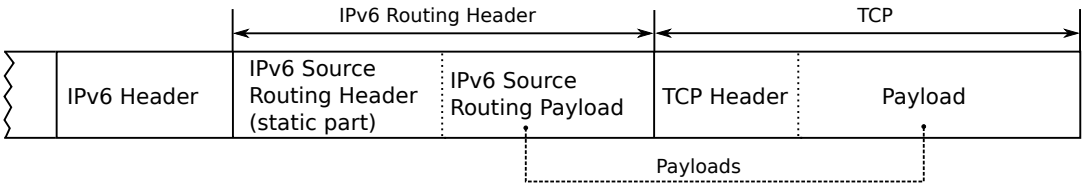


Figure 3.5: The example of packet with more payloads.

followed by IPv6 routing header and TCP header. In this case, we need to extract payload from the IPv6 routing header because of its variable length (the length is signaled in corresponding header field). Finally, the *parser* extracts payload from last available header which is the TCP. The example of this situation is shown in Fig. 3.5.

The *Match+Action group* is a fundamental element for running of user defined operations on parsed headers. It logically unifies implemented functionality like VLAN tagging, network address translation, and so on. Each *Match+Action group* contains one or more *Match+Action routers* and *Match+Action tables* which are connected to processing pipeline for implementation of required functionality.

Match+Action table maps incoming parsed headers and metadata to user defined actions which are performed in the same module. Selected action in *Match+Action table* depends on rules which are uploaded by configuration tool into *Match+Action tables* at runtime. For example, we can upload a rule which performs network address translation for specific addresses in given subnet, modification of parsed headers (or metedata fields), and so on. Each *Match+Action table* also computes the address of next table or router in pipeline. The selection process of next table or router can be conditional (i.e., address is identified from actual header values, selected action, hit/miss result in match table, and so on) or unconditional (i.e., table doesn't need any additional information for computation of next address). Therefore, each *Match+Action table* performs three operations: (1) search for most suitable action based on metadata and parsed headers, (2) identification of next table or router address and (3) invocation of selected action.

Match+Action router is the lightweight *Match+Action table* which is used for identification of next table or router address (i.e., it doesn't contain any match or action engine). It is typically used for implementation of P4's *if-else* selection statements. Generally, the selection is based on implemented logic in this module (e.g., constant, arithmetic equation, and so on). If actual Match+Action processing element (table or router) is not addressed, all incoming data are passed through the block untouched. We provide detailed description about *Match+Action router* and *Match+Action table* in Chapter 5.

All (possibly modified) headers are passed to the *checksum updater* which updates protocol checksum fields. The reason for this is that values of protocol fields may have changed in Match+Action pipeline which leads to computation of new checksum value. Finally, the output packet is constructed from incoming headers and payloads in *deparser* block. This block also performs possible discarding of packets (based on special signal in control path) because we need to remove all corresponding data from the proposed pipeline. The last deparser's output are metadata fields from Match+Action pipeline. We provide this data structure to other possible algorithms and modules after the deparser. The example of such algorithms can be QoS, other advanced filtering, sending of deparsed packets to output port or software tool, and so on.

The proposed pipeline is similar to the architecture introduced by Gibb [56]. However, our architecture (from Fig. 3.6) is more general because it supports protocols with more payloads and we also consider future utilization of HLS for fast implementation of user defined actions. The architecture of our processing pipeline is generally defined for implementation of different network applications. It is based on connection of all modules by

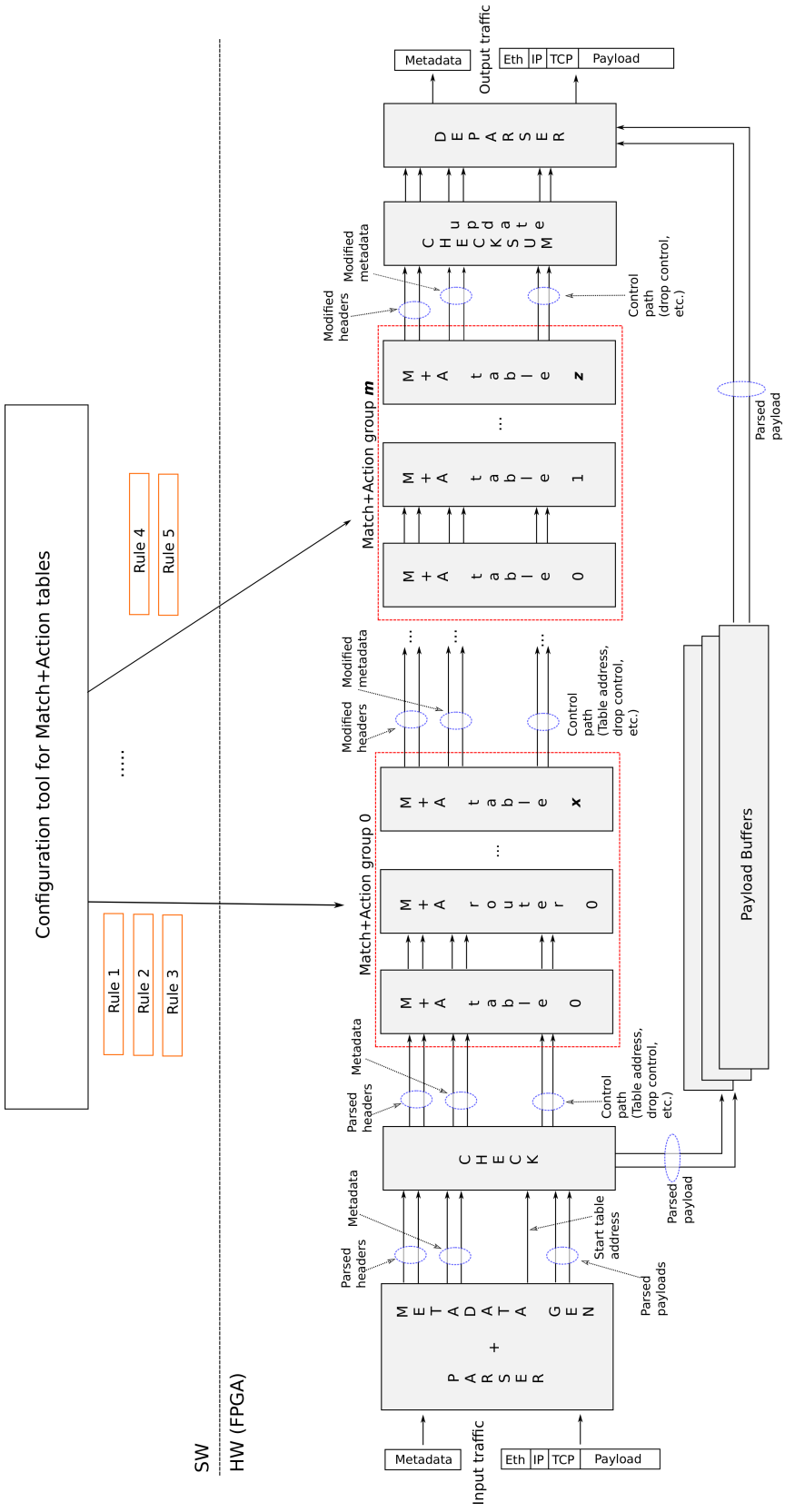


Figure 3.6: Architecture of high-speed processing pipeline.

unified communication interface. The unified interface consists of:

1. **Header Path** is generated from scratch because this communication line depends on set of supported protocols.

2. **Metadata Path** can be divided into two subtypes — static and dynamic. The static metadata are always present in each implemented application. Structure of dynamic metadata depends on user preferences because each implemented use case requires different user defined structures. The text provides more details in following text.

3. **Control Path** is a static structure which is part of each implemented use case. The main task of this interface is to control the processing in proposed pipeline. Namely, it controls the selection of next *Match+Action table* or *Match+Action router* address, discarding of processed packet and propagation of possible errors.

3.3 Mapping P4 Language to Processing Pipeline

The following section briefly introduces the idea of mapping from P4 language constructs to our architecture. The P4 language is an open standard which is defined by the P4 Language Consortium in [2]. We briefly introduced this language in Sec. 2.3.5, including basic constructs and main ideas. As we noticed, the P4 language defines five aspects of packet processing:

1. **Header Format** describes protocol headers and metadata recognized by the device. This group also contains definition of intrinsic metadata which are part of P4 language specification (i.e., they are part of each P4 program)

2. **Packet Parser** describes the (conceptual) state machine used to traverse packet headers from start to end, extracting field values as it goes.

3. **Table Specification** defines how the extracted header fields are matched in possibly multiple lookup tables.

4. **Action Specification** defines compound actions that may be executed for packets.

5. **Control Program** puts all the above together, defining the control flow mainly among the tables.

Tab. 3.1 briefly introduces the idea for mapping of P4 program to proposed architecture. We will provide more details about each group in following text.

3.3.1 Mapping to Metadata, Header and Control Path

This section introduces mapping of P4 program to metadata, header and control path. Each program contains specification of supported protocol headers. We provide examples

Pipeline block	**P4 language aspects**
Parser	Header Formats, Packet Parser
Metadata generator	Header Formats, Packet Parser
Check	Header Formats
Definition of M+A group	Control Program
M+A table	Table Specification, Action Specification
M+A router	Control Program (*if-else* statements)
Ordering of M+A tables in M+A group	Control Program structure
Checksum update	Header Formats
Deparser	Header Formats, Packet Parser
Metadata path	Header Formats
Header path	Header Formats
Control path	Static for each device: ◦ M+A table or router address ◦ Error bus ◦ Data flow control (source/ready signals, packet drop, etc.)

Table 3.1: Basic mapping of P4 language aspects to high-speed pipeline blocks; M+A is abbreviation for Match+Action.

of P4 source as a reminder of this language in all corresponding sections. The header and metadata specification has the same syntax and it looks like this:

```
header ethernet {
   fields {
      dst_addr  : 48; // width in bits
      src_addr  : 48;
      ethertype : 16;
   }
}
```

The proposed declaration introduces the example of Ethernet header. Each specification contains declaration of field name followed by width in bits. P4 program typically uses header references as the lowest granularity for managing of protocol fields, selection of next used table, and so on. We also need to detect the availability of parsed protocol in actual processed packet. Therefore, each protocol header is represented by following signals:

- **Data bus** for parsed protocol fields. The width of allocated bus is equal to the sum of all defined protocol fields.
- **Validity signal** for indication of protocol availability in actual processed packet.

The Header Format specification also supports protocols with variable protocol field length. This situation is similar as before because protocol payloads are parsed out and moved to stand alone Payload Buffers and the rest of remaining header fields has the static length (known during compilation of P4 program).

The similar situation is in the case of metadata path. The only difference is related to validity signal. We don't need to signalize the validity of metadata path values because metadata fields are associated to actually processed packet and we consider them to be always valid.

The control path is used for controlling of data flow across the high-speed processing pipeline. It consists of following signals: (1) identification of next table or router address, (2) signalization of errors during packet processing (e.g., signalization of processing errors during execution of action in Match+Action table, and so on) and (3) signalization for packet flow control. The packet flow control includes signals for controlling of header and metadata bus transfers (source/destination ready signals) and drop control at the end of processing pipeline in deparser.

3.3.2 Mapping to Parser, Metadata Generator and Check

This section introduces the idea for mapping of P4 program to *parser* and *metadata generator*. To achieve this task, we need to use two aspects of P4 language — Header Formats and Packet Parser specification. The first aspect, Header Formats, defines the structure of supported protocols. Therefore, we are able to assign extracted data vector to header fields. We introduced details about Header Format specification in previous section.

The second aspect, Packet Parser specification, is used for the description of relations between protocol headers. To achieve it, we can define a conceptual finite state machine which is used for extraction of incoming data. The state for processing of Ethernet header looks like this:

```
header ethernet eth;
parser parse_ethernet {
   extract(eth);
   switch(eth.ethertype) {
      case 0x8100: vlan;
      case 0x9100: vlan;
      case 0x0800: ipv4;
      default : ingress;
   }
}
```

This P4's language construct was briefly introduced in Sec. 2.3.5. Simply said, the *parser* statement defines one state of parse graph. The *switch* statement defines transitions to next states (*vlan* or *ipv4* in our example) or control programs (*ingress* in the code above). Graphical representation of the example is in Fig. 3.7. Introduced behavior needs to be implemented in processing pipeline. The architecture of our parser is based on

HFE M2 which was introduced in [58]. We provide more details about architecture and transformation process in Sec. 4.1.

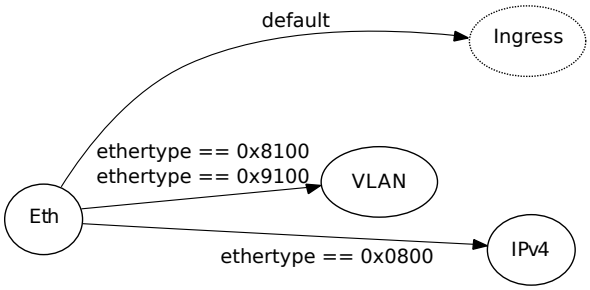


Figure 3.7: Graphical representation of *parse_ethernet*.

The second block, *metadata generator*, is used for generation of metadata field values for each processed packet. As we proposed before, metadata structure is inferred from Protocol Header declaration because metadata are syntactically defined in the same form. Initial values of metadata fields are inferred from Packet Parsed definition using the *set_metadata* statement. Therefore, the *metadata generator* generates the initial values of all required metadata structures as described in P4 program.

Mapping to *check* module is also quite straightforward and intuitive. The P4 program allows to define the *field_list_calculation* statement which simply binds checksum algorithm to the list of input fields. This list is provided in *field_list* statement. The following example was taken from [2]:

```
field_list l3_hash_fields {
    ipv4.srcAddr;
    ipv4.dstAddr;
    ipv4.protocol;
    ipv4.protocol;
    tcp.sport;
    tcp.dport;
}

field_list_calculation ecmp_hash {
    input {
        l3_hash_fields;
    }
    algorithm    : crc16;
    output_width : 16;
}
```

The proposed example introduces declaration of L3 hashing, including *ipv4* and *tcp* protocol fields. The *check* module setups initial values of control path signals based on result from this stage. In the case of any error, the processed packet is marked as invalid and

the next processing depends on implemented behavior of Match+Action pipeline elements. The general implementation of the *check* module is part of our ongoing research.

3.3.3 Mapping to Match+Action Group

This section introduces the idea for mapping of P4 program to Match+Action processing pipeline. The fundamental processing unit of our pipeline is *Match+Action group* which logically unifies complex operations like ACL, packet filtering, and so on. Some complex programs are divided into more steps because the next step of implemented algorithm depends on the result of previous. The P4 language allows to define complex processing program in *control program* statement. Each program contains one or more conditional statements (*Match+Action routers*) or table calls (*Match+Action tables*) which are connected into deep pipeline. Each *Match+Action table* has three functionalities: (1) mapping of headers and metadata fields to most suitable action, (2) modification of protocol headers and metadata fields based on selected action and (3) selection of next Match+Action element in processing pipeline. The P4 language allows to define the structure of *Match+Action table* using the *table* statement. Each P4's table statement contains two declarations: (1) *reads* statement which defines the structure of search key (i.e., it defines a list of used protocols/metadata fields together with matching algorithms) and (2) *actions* statement which defines a list of allowed actions. The simple filtering in P4 language looks like this:

```
// Filter table.
table filter {
    reads {
        ipv4.srcAddr : lpm;
    }
    actions {
        _permit;
    }
}

//Drop table (no read block => just run the action)
table drop {
     actions{
        _drop;
    }
}

// Filtering program
control ingress {
    // Drop all non-IPv4 traffic
    if(valid(ipv4)) {
        // Try to find the rule
        apply(filter) {
           //Table miss  = drop packet
```

```
            miss {apply(drop);}
        }
    } else {
        // non-IPv4 traffic has been detected
        apply(drop);
    }
}
```

The proposed example introduces two tables — *filter* and *drop*. The first table, *filter*, defines the used protocol field and possible action (*_permit* in our case). The second table, *drop*, defines one possible action (*_drop)* and no matching field. In such case, the P4 defines to use the default action. The *control program* defines the order of Match+Action elements in the processing pipeline. In our case, we start with evaluation of *if-else* statement which drops all non-IPv4 traffic by execution of drop table. The conditional statement is mapped to *Match+Action router*. Each IPv4 packet is passed to the *filter* table which is followed by the *drop* table in the case of table miss (in *filter*). The text provides two graphs. The Fig. 3.8 represents the graph of control program and Fig. 3.9 shows the structure of Match+Action pipeline.

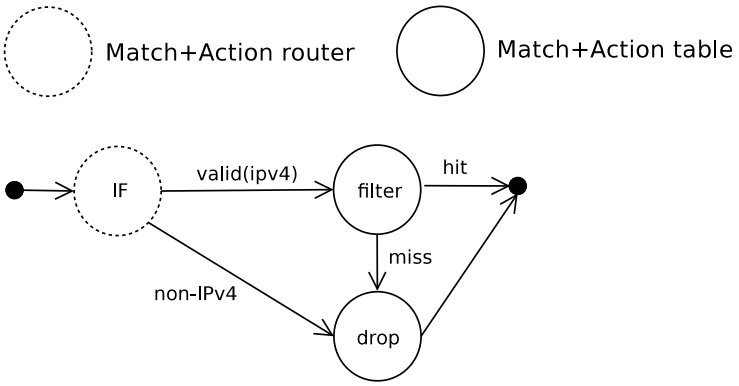


Figure 3.8: Graph of *ingress* control program.

The proposed example demonstrates the main idea for mapping from P4 language to pipeline of Match+Action elements. That is, *control program* defines the *Match+Action group*, all tables in a group are mapped to *Match+Action tables* and all conditional statements are mapped to *Match+Action router*. We provide more details about the transformation of P4 program to Match+Action element pipeline in Chapter 5.

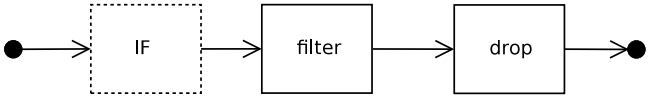


Figure 3.9: Match+Action pipeline structure of *ingress* control program.

3.3.4 Mapping to Checksum Update and Deparser

This section demonstrates the idea for mapping from P4 program to *checksum update* and *deparser* module. Firstly, we start with introduction of *checksum update*. The idea for mapping is similar to mapping of P4 program to *check* module (see Sec. 3.3.2 for more details). This module also uses P4's *field_list_calculation* statement for update of hash fields because some data may have changed in Match+Action processing pipeline. The general structure of this module is part of our ongoing research.

The last module, *deparser*, is the output module in proposed pipeline architecture. The main function of this module is to construct the output packet from incoming header fields. *Deparser*'s behavior is inferred from P4's Header Format and Packet Parser definition because deparser performs inverse operation to parsing. We don't discard any metadata from the processing pipeline and we transfer all fields to the output together with constructed packet. These metadata fields are used by other blocks after the deparser. We provide more details about the transformation of P4 program to *deparser* in Sec. 4.2.

3.4 Summary

The chapter introduces target architectures for mapping of P4 program. Firstly, the text introduces not only state-of-the-art software based targets, but also more powerful hardware targets including NPUs and FPGAs. We selected the FPGA as our target architecture because it connects the flexibility of software with parallel nature of hardware into one compact package (i.e., the functionality of the hardware can be reprogrammed).

Secondly, we introduced the architecture of our high-speed processing pipeline (published in [A.7]). The main idea of the architecture is based on the Parser-Deparser model (published in [A.10]) which is flexible and suitable for implementation of general packet processing engine in hardware. We demonstrate the flexibility of this approach on two common use cases — VLAN tagging and network traffic analysis. Finally, the last section introduces ideas for mapping of P4 language constructs to our high-speed pipeline architecture.

Chapter 4

Parser and Deparser Architecture

In this chapter, we introduce the architecture of parser and deparser blocks with further details about transformation algorithm and generated architecture. This chapter also introduces the experimental results for parser and deparser, including the time complexity analysis of transformation algorithms.

4.1 Parser Architecture

As we noticed before, the parsing process is defined by two P4's aspects (Header Formats and Packet Parser definition). The following text introduces the architecture of firmware parser called HFE M2 [58] and the transformation algorithm from a P4 description to the HDL code (published in [A.1, A.2, A.12, A.8]). Finally, we describe how to infer special parameters of parser which allow to reduce FPGA resources of generated engines.

4.1.1 Overview of HFE M2 Architecture

The main idea of automatic generation of 100 Gbps parsers comes from the architecture of HFE M2 which was introduced by Puš et al. in [58]. The paper presents a block architecture of hand-written modules which is capable to process traffic at speed of 100 Gbps. However, it doesn't present any algorithm for mapping from abstract description to proposed architecture. Therefore, we adopt the architecture of HFE M2, identify fundamental types of blocks in Protocol Analyzer structure (see later), and provide transformation process from P4 language to synthesizable implementation.

The HFE M2 architecture consists of two main block types — Protocol Analyzers and pipelines. A generic interface is used for connection between the protocol analyzers. There is an optional pipeline block between each two Protocol Analyzers. The pipeline blocks can be individually enabled/disabled at compile time to tune the final frequency, latency and chip area. Protocol analyzers and pipeline blocks are connected to the processing chain which represents the protocol stack of incoming network packet. An example of the HFE M2 processing chain is shown in Fig. 4.1.

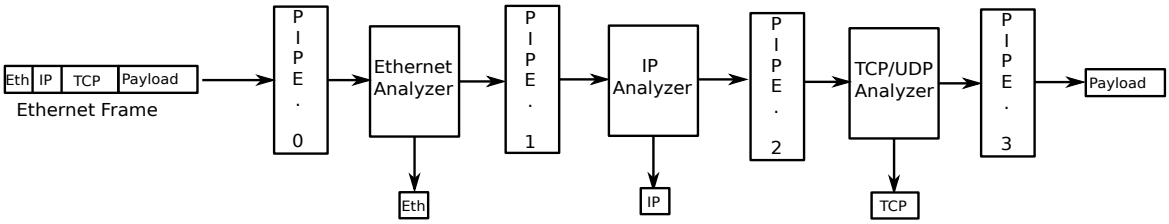


Figure 4.1: HFE M2 architecture.

Protocol analyzer uses the *Generic Protocol Parser Interface* (GPPI) for connection between modules. This interface provides the input information necessary to parse a single protocol header. That is: (1) current packet data being transferred at the data bus, (2) current header offset within the packet and (3) expected protocol to parse. GPPI output information includes (4) extracted packet header field values, plus the information needed to parse the next protocol header: (5) next header offset and (6) type of the next protocol header. More details about the GPPI can be found in [58]. Brief architecture of each Protocol Analyzer block is shown in Fig. 4.2.

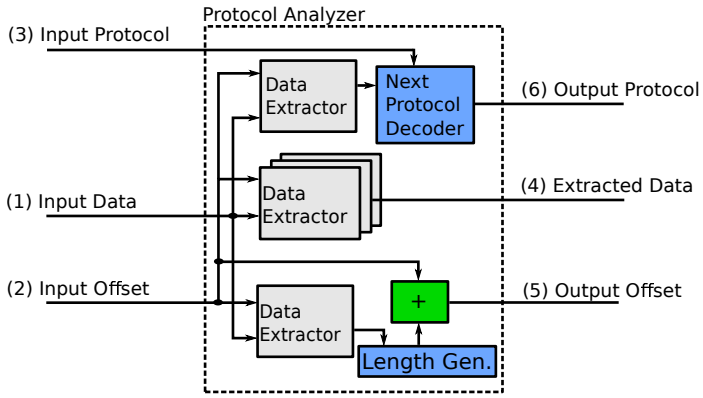


Figure 4.2: Protocol analyzer architecture.

Protocol analyzer architecture contains four block types: (1) Data Extractors, (2) Next Protocol Decoder, (3) Length Generator and (4) Adder. Data Extractors are used to extract packet data from a given offset. Data Extractors are configured with two parameters: *Extract Length* and *Extract Offset*. The first parameter defines the number of extracted bytes from packet data. The offset of data within the packet is computed as the sum of current header offset (a value from input GPPI interface) and the Extract Offset parameter. Note that Data Extractor blocks contain multiplexers which allow to extract data from any byte position. These multiplexers can be configured with some additional optimization parameters which reduce consumed FPGA resources. We describe these parameters in Sec. 4.1.3.

Next Protocol Decoder is used to compute the next expected protocol. Its structure fully depends on the protocol header format. Generally, it is a function converting some extracted packet header bytes into an internal number representing the protocol type.

Length Generator block is used to compute the length of current protocol header, so that it can be added to the Input Offset signal to obtain the Output Offset signal, which represents the start offset of the next protocol header. The added offset value can be a constant or a result of an expression (see the header format specification in previous chapter).

From the perspective of parser generation, we can identify three types of blocks in the Protocol Analyzer structure (see Fig. 4.2): (1) Static (green color), (2) Configured (grey color), (3) Fully protocol-specific (blue color). The static block is used in every Protocol Analyzer without any change. The Protocol Analyzer architecture contains only one static block, which is the adder. This block is used to compute the next protocol offset from current Input Offset and Length Generator output. The second group of blocks is general enough for usage in all Protocol Analyzers, only with different parameters settings. The architecture contains several Data Extractor blocks which are instantiated (from a hand optimized template) and configured regarding to P4's Header Format specification. Blocks marked by blue color are entirely protocol specific, so that every Protocol Analyzer needs custom implementation of them. This means that Protocol Decoder and Offset Generator must be uniquely generated for each Protocol Analyzer from P4's Header description.

This architecture of Protocol Analyzer is general enough for processing of most L2-L4 protocols. For this target block structure, we can generate, configure and connect all described blocks in automatic way from a P4 program. Details of this transformation process are discussed in the next section.

4.1.2 Transformation from the P4 to Parser Architecture

Transformation algorithm is one of the key problems addressed in this chapter. As we note in Sec. 4.1.1, HFE M2 architecture consists of Protocol Analyzers and pipeline modules which are connected to form the processing chain. The transformation from P4 to parser can be divided into two problems — (1) Generating the Protocol Analyzers and (2) Generating the processing chain. Inputs of the transformation process are P4's Header Format and *Parser Graph Representation*.

We define the *Parser Graph Representation* (PGR) as an acyclic oriented graph which is generated from the P4's Packet Parser definition. Each node (or state) represents one packet header and each transition represents the next parsed protocol header. Each transition is taken based on the parsed data. Condition of a transition is inferred from the P4's Packet Parser definition. Each non-final state contains additional transition to the *Unknown* state. This state is not explicitly described in P4 program but it is implicitly required by the parser. It represents the situation when no value matches the actual set of transition conditions (i.e., we cannot continue in parsing of next protocol header). Each PGR node also contains a pointer to P4's Header Format definition which is needed during generation of individual Protocol Analyzers. The PGR representation is built from a P4's Packet Parser definition. We introduce more details about this structure in the following text. An example of this generated representation is in Fig. 4.3. The figure doesn't show transition conditions in order to keep it well arranged.

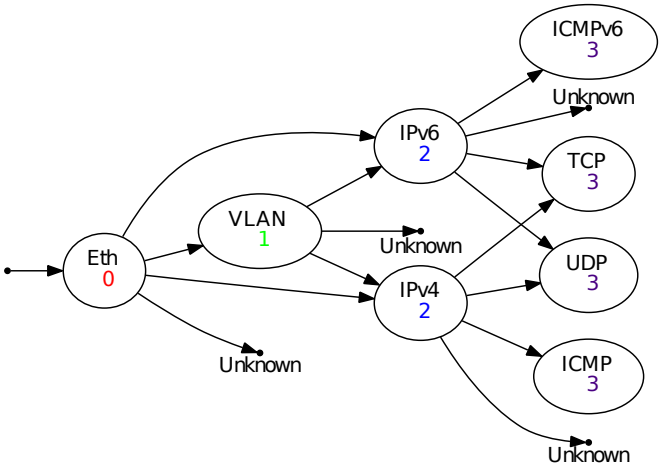


Figure 4.3: Parser Graph Representation; The example supports Ethernet, VLAN, IPv4, IPv6, TCP, UDP, ICMP and ICMPv6.

As we note in 4.1.1, each Protocol Analyzer consists of three types of blocks (see Fig. 4.2). We now describe the generation of Protocol Analyzer block:

- The Length Generator block is derived directly from P4's Header Format definition. It can be either a constant in the case of constant length header, or a (usually simple) formula in the case of variable length header.
- The Next Protocol Decoder is also generated from the P4's Packet Parser description. Each transition from the parser state is described in the P4's *switch* statement by the tuple, including *value* and *next state*. Therefore, we can implement Protocol Decoder by a multiplexer which selects the next protocol based on currently parsed values. The protocol headers that follow the currently parsed one are found in PGR during the generation. Data Extractor blocks are parameterizable modules which are used in all Protocol Analyzers without any change, only by setting the parameters to match the target protocol. There are two parameters for each Data Extractor: Extract Length (number of bytes to be extracted) and Extract Offset (byte position of the extracted field, relative to the first byte of protocol header).
- Extracted Data of Packet Analyzer can be inferred from the Header Format specification because we know the structure of protocol fields in the parsed protocol. Therefore, protocol fields can be extracted from packet data using the list of protocol fields and their sizes.
- Adder is a static block common to all Protocol Analyzers.

When generating the Next Protocol Decoder, Length Generator blocks and Extracted Data outputs, Data Extractors are instantiated and parametrized as needed. Both parameters (Extract Length and Extract Offset) are directly derived from the P4 description of translated program.

Algorithm 1: Recursive algorithm for identification of node levels.

```
Function FindNodeLevels(node, curr_level)
    Input: node = actual node to process
    Input: curr_level = actual level of the node
    Result: Node with updated maximal level
    begin
        if node.fresh == False then
            return;
        end
        /* Mark the node as not fresh and update the level                */
        node.fresh = False;
        act_level = node.get_level();
        if act_level <curr_level then
            node.set_level(curr_level);
        end
        /* For all fresh successors, update the level and call the same
           function                                                        */
        node_successors = node.get_next_states();
        for next_node in node_successors do
            /* Don't call the node if the longest path already exists     */
            if next_node.get_level() - node.get_level() <1 then
                FindNodeLevels(next_node,curr_level+1);
            end
        end
        /* Mark the node as not visited                                   */
        node.fresh = True;
        return;
    end
```

Previous text introduces the automatic generation of Protocol Analyzer from P4 source code. The following text provides details about the automatic generation of processing chain. The key problem is to identify a place for insertion of each Protocol Analyzer in the chain. Therefore, the planning algorithm has to fulfill following requirements:

1. The planned structure of modules has to form a pipeline (due to the parser architecture).

2. Protocol at the end of each transition has to be processed after the protocol at the beginning of that transition.

3. The ordering of Protocol Analyzers has to be carefully planned because the architecture doesn't allow feedback loops. Planned modules have to form a totally ordered set such that it contains all possible ways through the PGR (node skipping is allowed).

Algorithm 2: Brief transformation algorithm from P4 to parser.

```
Procedure TransformationToParser(prog)
    Input: prog = P4 Program
    Result: VHDL code of the parser
    begin
        /* 1) Identify the Parser Graph Representation                    */
        parser_graph = GetParserGraphRepresentation(prog);
        /* 2) Mark all nodes as Fresh.  After that, traverse through the
              graph and identify level of each node                       */
        MarkFresh(parser_graph);
        FindNodeLevels(parser_graph.root,0);
        /* 3) Generate Protocol Analyzers and processing chain            */
        GenerateProcessingChain(parser_graph);
    end
```

4. The ordered set can be found using the *depth-first search* (DFS) algorithm where the output is a *level* of each node (i.e., the latest possible use of a protocol in pipeline).

The generator of processing chain uses a PGR as an input. Its task is to identify the longest paths from root to each node in PGR (i.e., *level* of each node). If we have several nodes on the same *level*, we connect Protocol Analyzers in series with arbitrary ordering. While same-*level* analyzers could be connected in parallel, the serial approach allows us to keep the homogeneous structure of processing chain. The longest paths in a PGR is found using the Alg. 1. The algorithm recursively traverses and identifies node *levels* in inspected graph. The result of this algorithm is shown in Fig. 4.3, where each node contains a number which represents the length of the longest path from root.

Finally, we introduce the Alg. 2 which is used for generation of complete parser architecture from a P4 description. We implemented this transformation algorithm in Python language with usage of P4-HLIR [46] project. The result of Alg. 2 can be seen in Fig. 4.4 which represents the processing chain generated from the PGR in Fig. 4.3. The Fig. 4.4 doesn't contain any pipeline modules for brevity, but the real firmware implementation

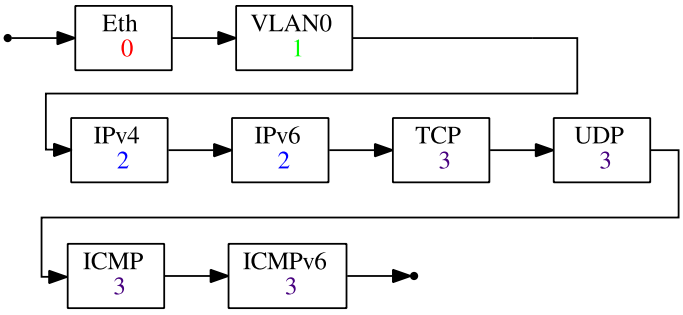


Figure 4.4: Generated processing chain; Pipeline modules are omitted for brevity.

contains a pipeline module between each two adjacent Protocol Analyzers. This figure also shows the situation when two or more different nodes are situated on the same level. Such nodes are connected in series and their relative position doesn't matter. The only rule is to keep them together (i.e., the generator connects modules which belong to the same level in series).

4.1.3 Optimizations

The original hand-written HFE M2 parser supports several optimizations which save a significant amount of chip resources. Therefore, we have decided to support some of these optimizations in our generator as well.

The first optimization is related to Protocol Analyzer's GPPI interface. The key idea is to optimize the width of the offset bus which is used for signalization of protocol header start position (Input Offset and Output Offset signals). In another words, Input Offset and Output Offset signals are *pointers* into the packet. Since the protocol stack is being analyzed in sequential manner within the packet, the width of offset bus can increase in sequential manner too — it is unnecessary to implement all logic (adders, etc.) at full data width, especially in early pipeline stages. The bus width parameter is inferred from maximal protocol header lengths during translation. The bus width at each level is computed as a binary logarithm of the sum of all preceding protocol header lengths. Narrower offset bus leads to smaller modules for computation of next header offset and other required values. Therefore, we save chip resources and possibly raise the working frequency.

The second optimization is related to data extraction which is performed by multiplexers within Data Extractor blocks. Generally, each Data Extractor is able to extract data from any byte position in the packet bus data word. The multiplexer is controlled by current data bus offset and offset of the desired field within the header. However, given the fact that the packet header may start only at certain positions on the data bus, current offset can contain only values with the corresponding resolution. This resolution is computed from P4's Header Format and Packet Parser definition. Using these two specifications, we identify the Offset List which contains all possible starting positions of each analyzed header in the processing chain. This knowledge is built from a protocol header length and relations between protocol headers by simulation of data transfer on data bus. Computed lists are used for identification of required multiplexer's parameters. By making

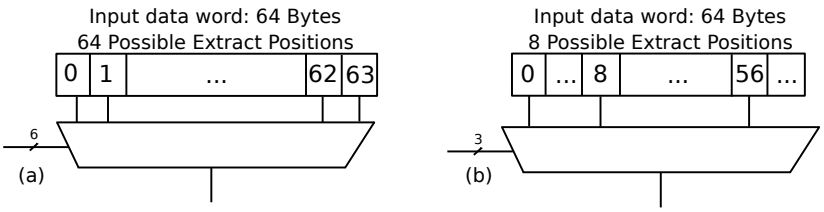


Figure 4.5: The example of data extraction multiplexer: full (a), optimized (b).

Data Extractors less general, we simplify the structure of each extraction multiplexer and save available chip resources. An example of this optimization is shown in Fig. 4.5.

The multiplexer parameters can be computed from Offset List $\mathbf{b} = (b_0, b_1, \ldots, b_n)$ using the Alg. 3 where each Offset List element represent one starting protocol offset. The proposed algorithm is based on observation that each offset can be expressed in the form $k * 2^n + q$ where:

- $k \in \mathbb{N}_0$ represents the index of multiplexer block,
- $2^n, n \in \mathbb{N}_0$ represents the size of multiplexer block,
- $q, 0 \leq q < 2^n, q \in \mathbb{N}_0$ represents the offset in multiplexer block.

In other words, we want to find a common granularity for all starting offsets which are represented by a multiplexer block size and internal offset.

There is also a place for optimizations of P4 program which cannot be automatically generated. Instead, it is required to optimize the program during design time. The generator can then benefit from more efficient input, which results in better design in terms

Algorithm 3: Computation of multiplexer parameters.

Input: Offset List: $\mathbf{b} = (b_0, b_1, \ldots, b_n), n \in \mathbb{N}_0$
Result: Computed parameters: $n \in \mathbb{N}_0, q \in \mathbb{N}_0$

```
/* Starting values (extraction from each byte position)                    */
n = 0;
q = 0;
div = 2;
while True do
    /* Compute the reminder for each element in tmp vector                 */
    for b_i in b do
        tmp_i = b_i mod div;
    end
    /* Check reminders in tmp vector                                       */
    if tmp_i == tmp_0, ∀tmp_i ∈ tmp then
        /* All reminders are identical.  Remember the actual result and try
           next iteration.                                                  */
        n = ⌈log_2(div)⌉;
        q = tmp_0;
        div = 2 * div;
    else
        /* All elements are not same.  Stop the algorithm.                 */
        break;
    end
end
return n, q
```

of latency and consumed resources. In this text, we introduce one such optimization of P4 program which leads to less Protocol Analyzer blocks being generated. The idea is to merge protocol headers that are compatible in terms of extracted protocol header fields. As an example, we want to extract source and destination ports of TCP and UDP protocols because their structure is similar. Therefore, we create a new custom protocol header which describes export of interesting fields. We don't have to worry about incomplete protocol header specification because our replaced protocol headers are leaves of the PGR (see Fig. 4.3), so that there is no further processing after this merged header. We define the new protocol header like this:

```
header tcp_udp_t {
   fields {
      src_port : 16; // width in bits
      dst_port : 16;
   }
}
```

4.1.4 Time Complexity of the Transformation

Time complexity of the proposed transformation Alg. 2 consists of following components (consider situation without computation of multiplexer parameters):

1. `GetParserGraphRepresentation`'s time complexity is equal to $\mathcal{O}(V + E)$ (DFS algorithm), where V is the number of nodes and E is the number of edges. Our transformation algorithm requires PGR with no cycles. In general, maximal number of edges in acyclic graph is equal to $\frac{n}{2} * (n - 1)$ where n is the number of Protocol Analyzers (i.e., nodes of our graph). Total time complexity of DFS is $\mathcal{O}(n + \frac{n}{2} * (n - 1)) \sim \mathcal{O}(n^2)$.

2. `FindLongestPaths`'s time complexity is equal to $\mathcal{O}(n^2)$ (DFS algorithm), where n is the number of Protocol Analyzers.

3. `MarkFresh`'s time complexity is equal to $\mathcal{O}(n)$, where n is a number of protocols (i.e., nodes of PGR).

4. `GenerateProcessingChain`'s time complexity is equal to $\mathcal{O}(n)$ because generated processing chain contains n Protocol Analyzers.

Total time complexity of the transformation (without computation of multiplexer parameters) is $\mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(2 * n^2) + \mathcal{O}(2 * n) \sim \mathcal{O}(n^2)$.

The list of protocol offsets is constructed using the DFS algorithm. Therefore, the time complexity is $\mathcal{O}(\frac{n}{2} * (n - 1))$ where n is the number of Protocol Analyzers. Notice that the length of Protocol Analyzers's offset list depends on complexity of PGR.

Computation of multiplexer parameters cannot use more than $\mathcal{O}(log_2(\max(\mathbf{b})))$ iterations (i.e., the number of binary digits of maximal element in the vector) where the

algorithm requires $\mathcal{O}(\text{len}(\mathbf{b}))$ time for initialization of *tmp* vector and $\mathcal{O}(\text{len}(\mathbf{b}))$ for check of all reminders in *tmp* vector. The total time complexity of the algorithm is $\mathcal{O}(2 * \text{len}(\mathbf{b}) * log_2(\max(\mathbf{b}))) = \mathcal{O}(\text{len}(\mathbf{b}) * log_2(\max(\mathbf{b})^2))$ where $\mathbf{b}$ is a merged list of protocol offsets of all Protocol Analyzers.

4.2 Deparser Architecture

The deparser module is used to assemble packets back from modified protocol headers. The architecture of our deparser module (published in [A.12, A.7, A.10]) is similar to HFE M2 [58]. It consists of two modules — Protocol Appenders and pipelines. Protocol Appender effectively inserts a protocol header before payload (i.e., it merges protocol header and payload). There is an optional pipeline block between each two Protocol Appenders. This pipeline block has the same reason as in the case of parser pipelines. They are also used for tuning of final frequency, latency and chip area. All modules are connected to the processing chain which represents the supported protocol stack. Brief architecture is shown in Fig. 4.6. The figure also introduces the main idea of deparser's architecture — all modules are connected in reverse order (from upper protocol layers to lower protocol layers), where each Protocol Appender inserts the header at zero offset. Generally, there are two approaches for stacking of appending units — botom-to-up and up-to-bottom.

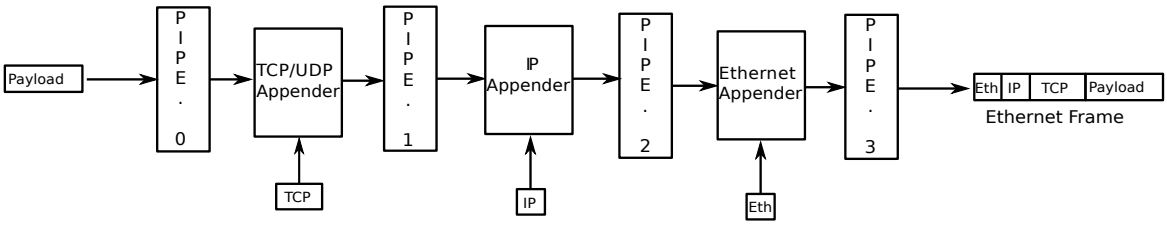


Figure 4.6: Deparser architecture.

The bottom-to-up approach is similar to the solution of parser (i.e., insertion from lower protocol layers to upper protocol layers). This approach is natural for the implementation in a software program which typically fills the packet memory in sequential manner. However, this is not suitable for hardware implementation because each Protocol Appender would have to support multiple *insertion offsets* (the set of supported insertion offsets depends on previously inserted headers). This leads to more complex shifting logic which consumes more FPGA resources.

The up-to-bottom approach makes the assembling process easier because the appended header is inserted always at the beginning of the packet (i.e., at zero offset). To make that possible, the incoming data of unfinished packets must be shifted to make space for the new header. The incoming data are fixed to zero offset. Therefore, the shifting logic is insensitive to previous Protocol Appenders and the complexity of shifting logic depends only on the current protocol.

The Protocol Appender uses a generic interface for connection between modules. This interface provides the input information necessary to assemble a single protocol header. That is: (1) current protocol header fields to be inserted, (2) current (unfinished) packet to insert the header into and (3) header insertion vector where each Protocol Appender observes one bit position. This observed bit enables or disables the header insertion. The output information includes (4) merged data and (5) unmodified header insertion vector. Notice that incoming unfinished packet always starts on the first byte of data bus.

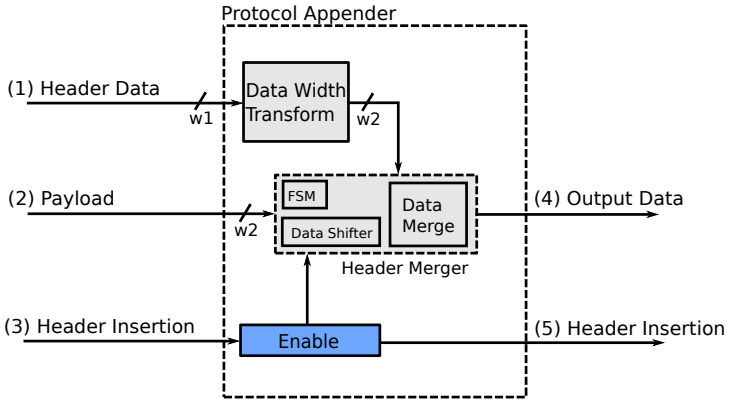


Figure 4.7: Protocol Appender architecture.

Architecture of Protocol Appender contains three block types: (1) Data Width Transform, (2) Header Merger and (3) Enable logic. The Data Width Transform unit is used to transform the header from its native data width (w1), given by the sum of its fields, to the width of the main data pipeline (w2). This transformation leads to a simpler data merging process. The Enable logic is a simple detector which observes predefined bit position in header insertion vector. The observed bit enables or disables the header insertion. The width of this input vector equals to the number of used Protocol Appenders in the processing chain.

The Header Merger consumes three inputs — transformed header data, payload and enable signal. It is divided into three sub-blocks. The first, Data Shifter, is used to make a free space in the incoming unfinished packet data. It moves the data behind the free space where the current packet header is to be inserted in Data merger sub-block. The Data merger sub-block implements an effective merging logic where both inputs (header and payload) are masked by logical *and* operation and merged by logical *or* operation. The whole process of merging is controlled by the Finite State Machine (FSM) which controls the logical operations. If no header is inserted in actual Protocol Appender stage, the FSM disables header interface (i.e., *and* operation with data and zero vector) and the payload is transferred unchanged (i.e., logical *or* between zero vector and incoming payload is performed).

There is also a possible optimization which can be inferred from the property of incoming protocol. All protocols can be divided into two groups — protocols with static and dynamic header length. In the first group, we have to support two shifting offsets

— zero offset (i.e., no change of payload structure) and static offset (i.e., offset of shifted payload by a constant). This kind of offset can be computed with equation 4.1 where l is the length of inserted header in bytes and w is the width of payload bus in bytes. In other words: payload is shifted to the next available byte position behind the inserted header. The proposed equation can be used if inserted header doesn't occupy the whole output bus width (the result of the equation 4.1 differs from zero). If header occupies the output bus (the result equals to zero), we have to move the payload insertion to the next bus cycle with zero offset value (i.e., we don't need to modify the offset of incoming payload). The example of header insertion is demonstrated in Fig. 4.8.

$$\text{offset} \equiv (l + 1) \pmod{w} \tag{4.1}$$

The case with dynamic protocol length is more complex, because we need to support more shifting offsets. The reason for this is straightforward — we don't know the exact length of the protocol header during the compile time. Therefore, we have to support more complex shifting logic based on granularity of inserted protocol. The shifting multiplexer can be also optimized in similar way like data extraction multiplexers in parser.

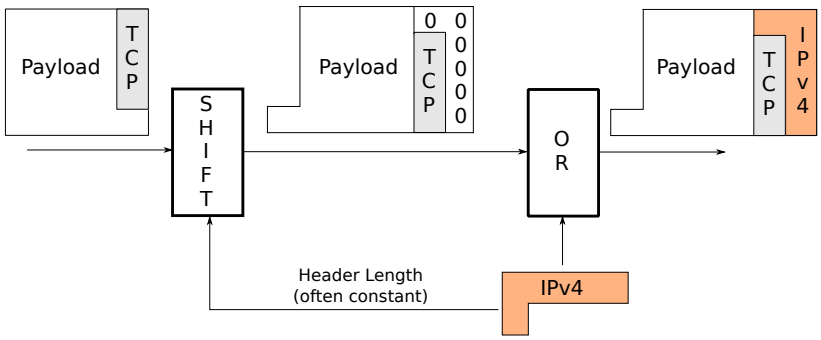


Figure 4.8: The example of header insertion in Protocol Appender.

4.2.1 Transformation from the P4 to Deparser Architecture

The transformation algorithm from P4 to deparser architecture is quite similar to generation of parser. We can reuse the PGR and *depth-first-search* (DFS) algorithm for identification of node levels. After the identification, we reverse the order of Protocol Appender modules. The transformation algorithm can be described using the Alg. 4

The result of the proposed algorithm is a processing chain similar to the example in Fig. 4.4. The only difference is the reversed order of all modules.

4.2.2 Time Complexity of the Transformation

The time complexity of the Alg. 4 is similar to the Alg. 2. However, there are two important differences. The first difference is related to `GenerateDeparsingChain`. This function

Algorithm 4: Brief transformation algorithm from P4 to deparser.

```
Procedure TransformationToDeparser(prog)
    Input: prog = P4 Program
    Result: VHDL code of the deparser
    begin
        /* 1) Identify the Parser Graph Representation                    */
        parser_graph = GetParserGraphRepresentation(prog);
        /* 2) Mark all nodes as Fresh.  After that, traverse through the
           graph and identify level of each node                          */
        MarkFresh(parser_graph);
        FindNodeLevels(parser_graph.root,0);
        /* 3) Reverse the order of PGR nodes and generate the processing
           chain                                                          */
        deparser_graph = ReverseNodeLevels(parser_graph);
        GenerateDeparsingChain(deparser_graph);
    end
```

contains computation of multiplexer parameters for appending logic. The logic needs to support just offsets of appended protocol. Therefore, the number of supported offsets is typically smaller than the list of supported offsets in the last Protocol Analyzer of more complex PGR. The second difference is related to the `ReverseNodeLevels`. This function is used for generation of deparser's structure in $\mathcal{O}(n)$ time where n is the number of Protocol Appenders. Therefore, the time complexity of the Alg. 4 is $\mathcal{O}(n^2)+\mathcal{O}(n^2)+\mathcal{O}(n)+\mathcal{O}(n)+\mathcal{O}(n) = \mathcal{O}(2*n^2)+\mathcal{O}(3*n) \sim \mathcal{O}(n^2)$.

4.3 Parser-Deparser Experiments

In this section, we introduce results for parser and deparser because both modules are usable standalone. We have tested properties of generated parsers with two different protocol stacks:

- **simple L2** - Ethernet, IPv4/IPv6 (with 2×extension headers), TCP/UDP, ICMP/ICMPv6
- **full** - Ethernet, 2×VLAN, 2×MPLS, IPv4/IPv6 (with 2×extension headers), TCP/UDP, ICMP/ICMPv6

For each protocol stack, we compare the manually optimized HFE M2 parser and the generated parser with all optimizations enabled. The deparser isn't compared to a hand optimized version because this architecture was tailored to needs of automatic generation and there is no other hardware based deparser available. All tested designs implement both engines without any additional logic like outputs FIFOs for parsed data, cleanup logic, etc.

We choose this approach because the additional logic typically depends on the character of implemented application. We use the Slice Logic (number of used LUTs plus FlipFlops) as a metric of resource utilization because these parts are the most utilized in most FPGA designs.

We provide results after synthesis for the Xilinx Virtex-7 XCVH580T FPGA using the Xilinx Vivado 2015.1 design tool. All designs were synthesized with different settings of data width and presence of pipeline modules. These settings, together with the resulting frequency, latency and resource usage, generate a large space of possible solutions for each P4 program. These solutions were searched for Pareto set which allows us to pick the best-fitting solution for an application.

4.3.1 Results for the Parser

In each **hand optimized** design test case, we use two different data widths: 256 and 512 bits. For each data width, every possible placement of pipelines was tested: 2^9 possible combinations in the case of the full protocol stack and 2^5 combinations in the case of simple L2 protocol stack (because there are 9 and 5 configurable pipeline stages).

In each **generated** parser test case, we use two different data widths: 256 and 512 bits. For each simple L2 parser, every possible placement of pipelines was tested: 2^{10} possible combinations. In the case of full parser, there are 2^{14} different configurations. We have randomly selected 20% of all possible solutions. This approach allows us to briefly inspect properties of generated processing chain in a reasonable compile time.

We provide two graphs with Pareto sets: one showing Pareto sets optimized for throughput and chip area without any regard to latency, and second optimized for throughput and latency without any regard to FPGA resources. Before the graphs, we introduce results for optimizations which are described in Sec. 4.1.3 because they have an influence on latency and used resources. We define the following notation:

- **No optimizations (O0)** - no optimizations are used.
- **Offset optimization (O1)** - optimization of the offset width.
- **Offset + multiplexer optimization (O2)** - offset and multiplexer optimizations.
- **Optimized P4 program (O3)** - O0 version with effectively written P4 program (manual optimization).
- **All optimizations (O4)** - all proposed optimizations are used.

Resource utilization (values in parentheses represent the percentual usage of available FPGA's resources) and latency for parsers generated with different optimizations are shown in Tab. 4.1. The table shows the influence of proposed optimizations on similar hardware configurations with throughput around 100 Gbps.

Opt.	Pipes	Latency [ns]	Thr. [Gbps]	Slice LUT [-]	Slice Reg [-]
O0	8	39.8	102.7	25335 (6.98%)	5055 (0.69%)
O1	14	75.3	101.9	21477 (5.91%)	8930 (1.23%)
O2	8	46.1	100.0	10103 (2.78%)	5537 (0.76%)
O3	9	44.5	115.1	14270 (3.93%)	5427 (0.74%)
O4	7	40.7	100.7	8314 (2.29%)	4795 (0.66%)

Table 4.1: Comparison of different optimization methods with resource consumption for the Xilinx Virtex-7 XCVH580T FPGA.

For all the following results we use O2 optimization, since O3 and O4 require manual modifications to the original P4 program. We consider this kind of optimization to be highly non-standard.

For comparison of the achieved Pareto set results for different protocol stacks, we provide graphs in Fig. 4.9 (throughput and FPGA resources) and Fig. 4.10 (throughput and latency). The Pareto sets show the best achievable solutions for our parsers. From these figures, we can see that supported protocol stack can significantly change parameters of the parser in terms of FPGA resources and latency. We can also see that P4 based parsers are approximately two times worse than hand-written parsers in terms of latency and consumed resources.

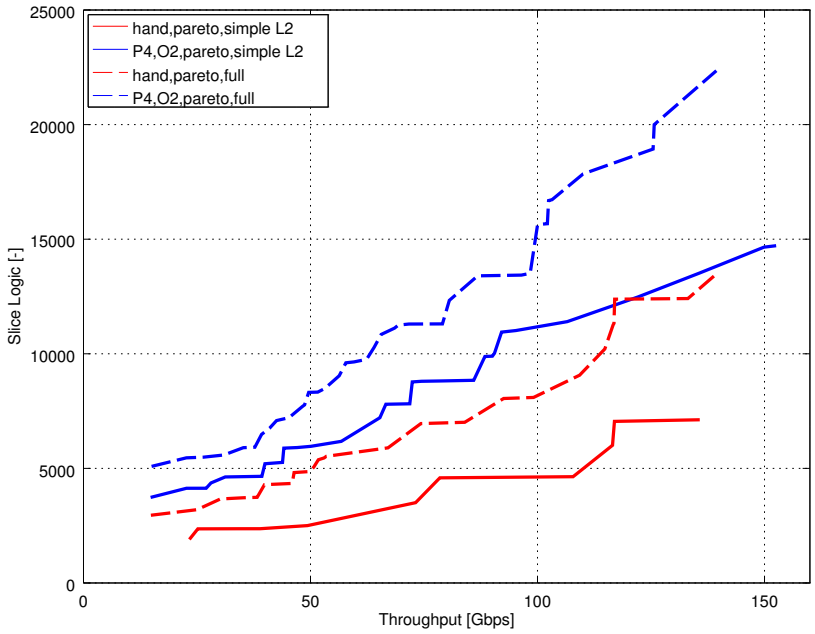


Figure 4.9: Parser - Comparison of the FPGA resource utilization versus throughput Pareto sets for the tested protocol stacks.

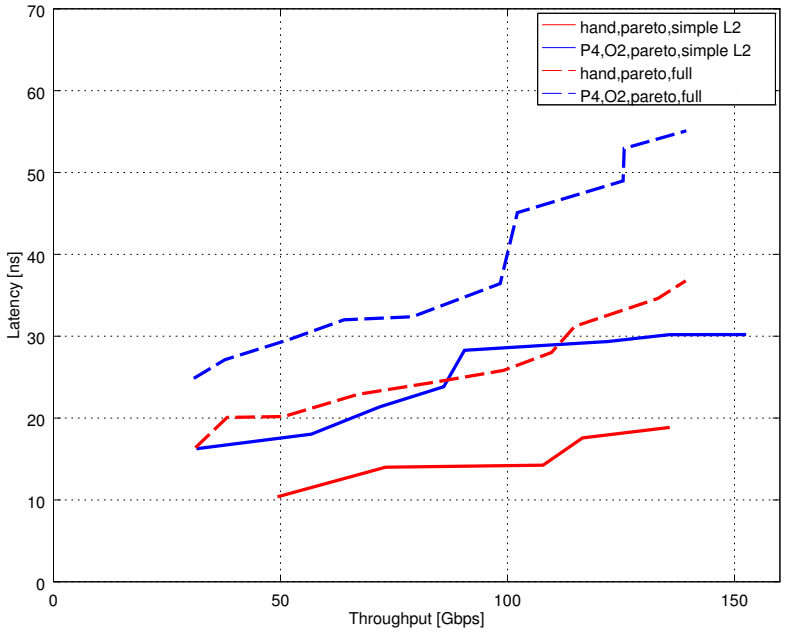


Figure 4.10: Parser - Comparison of the latency versus throughput Pareto sets for the tested protocol stacks.

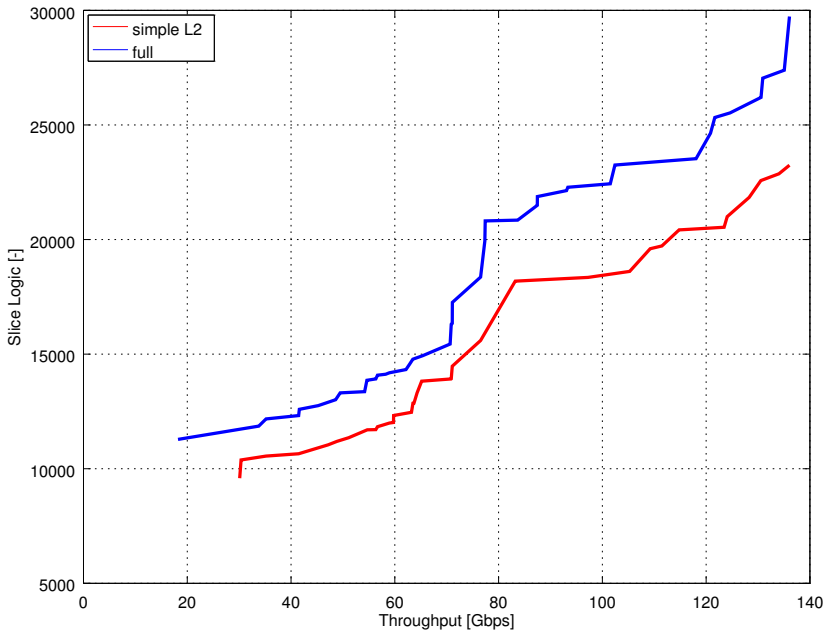


Figure 4.11: Deparser - Comparison of the FPGA resource utilization versus throughput Pareto sets for the tested protocol stacks.

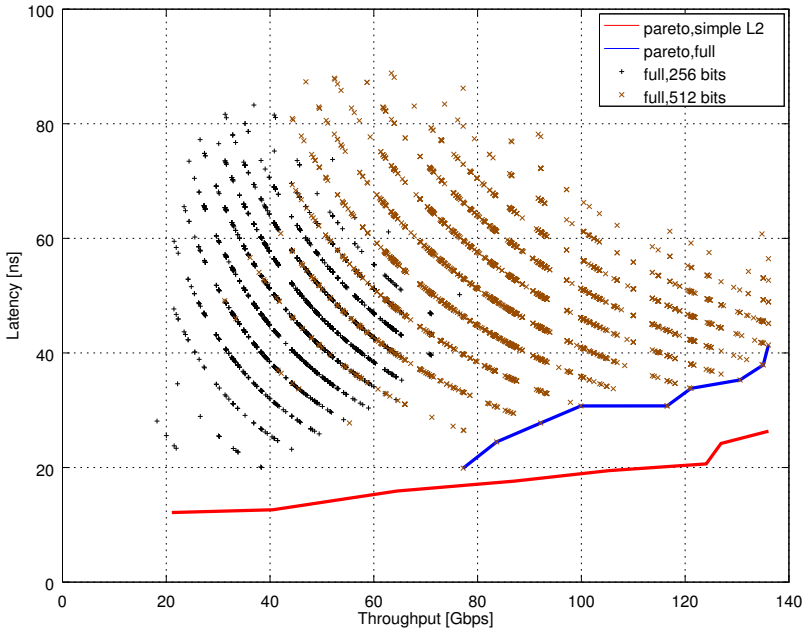


Figure 4.12: Deparser - Comparison of the latency versus throughput Pareto sets for the tested protocol stacks.

4.3.2 Results for Deparser

This section introduces results for the deparser module which is used for composition of output packet. We provide graph in Fig. 4.11 which introduces Pareto sets optimized for throughput and chip area. The graph also shows that consumed resources are higher compared to parser with the same throughput. The reason for this is the fact that each Protocol Appender contains payload shifting logic which consumes more resources than Data Extractor units in parser. The second graph, Fig. 4.12, introduces Pareto sets optimized for throughput and latency.

The experiment data were measured in the same way like experiments for **generated** parsers test cases. The generated test cases were chosen because we don't have any hand-optimized version of deparser which supports given protocol stacks (the architecture of deparser was developed with respect to automatic generation of target architecture). Pareto set for latency of full protocol stack starts approximately from 78 Gbps (see Fig. 4.12). We provide additional information for illustration of all measured data where each point represents a result for a specific configuration of deparser's pipeline.

4.3.3 Parser-Deparser Throughput

This section introduces the throughput analysis of Parser-Deparser blocks for the full protocol stack (see Sec. 4.3.1 for details). Two use cases will be introduced — No modi-

fication and VLAN tagging. The first, No modification, represents the raw throughput of both blocks where no changes are performed in parsed headers. The second, VLAN tagging, represents the throughput of the generated device during insertion of VLAN header. Both engines were generated and synthesized for COMBO-100G card [6] using the Xilinx Vivado 2015.4 design tool. The proposed device is equipped with the Xilinx Virtex-7 XCVH580T FPGA and is capable to process traffic up to 100 Gbps. All implemented designs were running at speed of 220 MHz. We provide two graphs: one showing the percentage throughput of 100 Gbps Ethernet line and second showing the number of mega packets per second (Mpps).

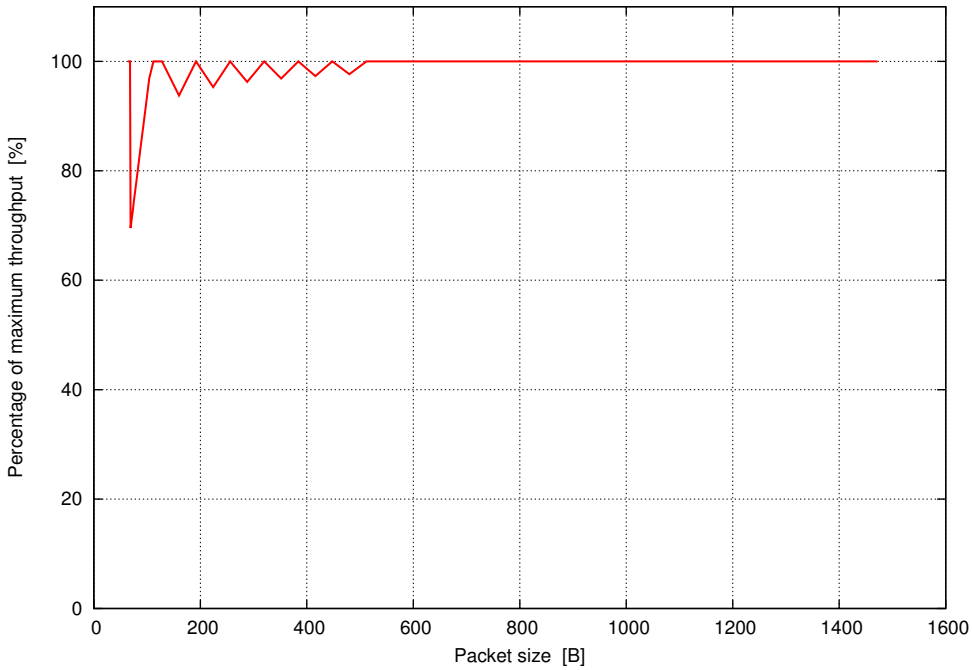
The Fig. 4.13 introduces results of raw throughput (no header modification or insertion was performed). We see that generated device is capable to process data at speed of 100 Gbps. However, there are some packet lengths where the throughput falls below 100 Gbps. This decrease of throughput is caused by ineffective usage of available output data bus in deparser. The most complicated situations occurs on short Ethernet frames. We demonstrate this problem on 512 bits wide data bus during transfer of 69 bytes (start of the frame is aligned to the first byte of data bus). In this situation, we need to use two bus cycles during which we transfer 69 bytes of Ethernet frame. Unfortunately, we are wasting available capacity (i.e., we use 69 bytes from 128 bytes available). The case gets less significant with longer Ethernet frames because the data bus is used for longer time and we need to reach smaller packet rate. Therefore, minimal throughput of aligned transfer on given packet length can be roughly computed using the equation 4.2 where dw is width of data bus, $freq$ is working frequency, pkt_len is length of a packet in bytes and min_cycles is the required number of bus cycles for transfer of the packet.

The situation for the VLAN tagging use case is introduced in Fig. 4.14. There is one additional source of decrease of throughput: inserting the VLAN tag makes the output packet 4 bytes longer. Therefore, the rate of packets decreases which leads smaller overall throughput (when measured at input).

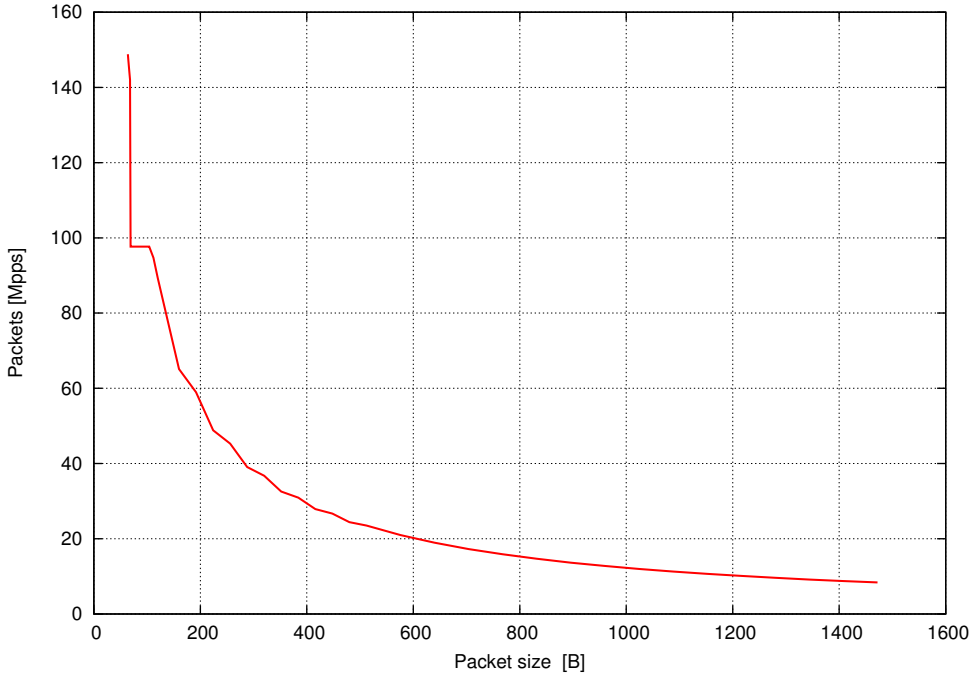
$$\begin{aligned} min\_throughput =& (dw * freq) * \frac{pkt\_len * 8}{min\_cycles * dw} && (4.2)\\ =& freq * \frac{pkt\_len * 8}{min\_cycles} && [bps] \end{aligned}$$

$$min\_cycles = \left\lceil \frac{pkt\_len * 8}{dw} \right\rceil \qquad [-]$$

The ineffective transfer can be solved by following idea — we need to generate and merge more network streams to one output line (we need to prepare required number of packets and merge them on output bus). Such solution will be capable to eliminate ineffective data transfer in deparser and reach the full output packet rate. Overview of this solution is shown in Fig. 4.15. We presented the approach for effective merging of network streams (architecture of *Merge* block) in [A.3].

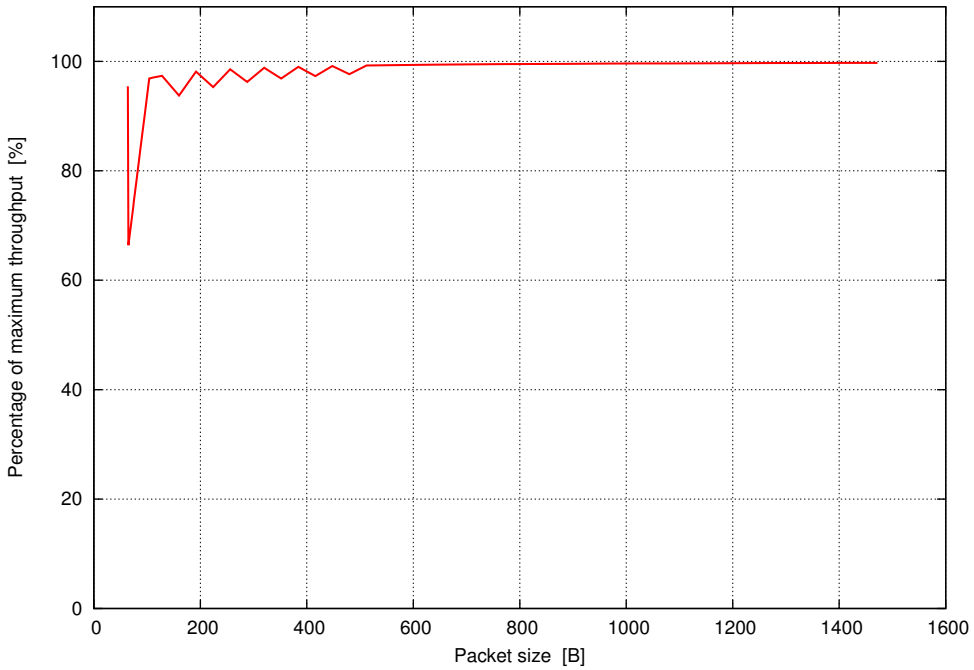


(a) Percentage of maximum throughput.

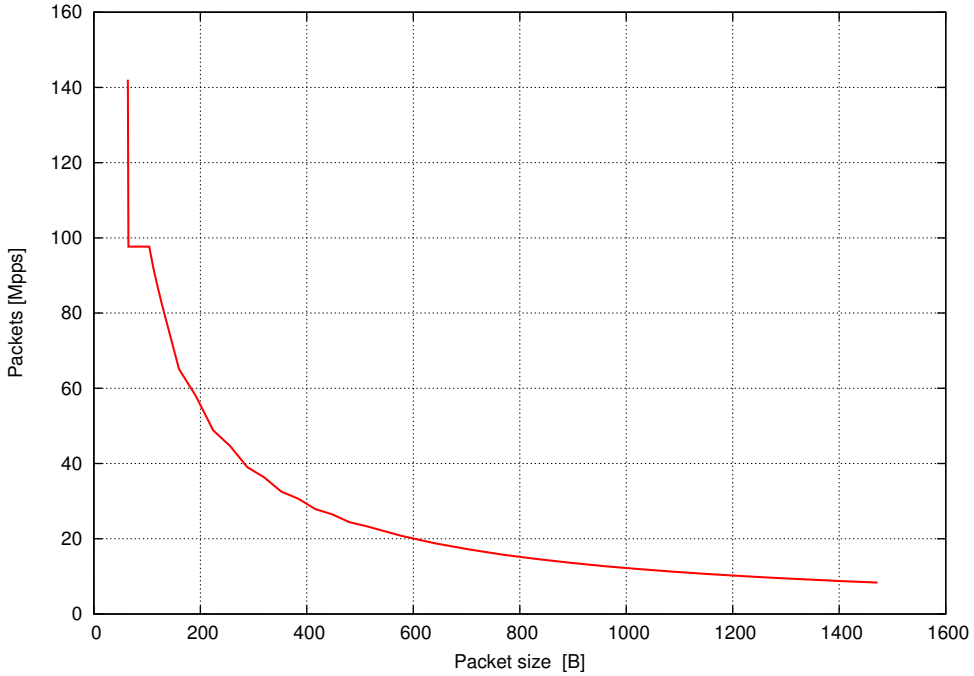


(b) Packet rate.

Figure 4.13: Throughput of generated Parser-Deparser device for the No modification use case.



(a) Percentage of maximum throughput.



(b) Packet rate.

Figure 4.14: Throughput of generated Parser-Deparser device for the VLAN tagging use case.

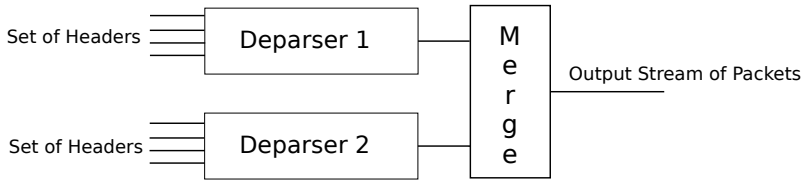


Figure 4.15: Idea for elimination of ineffective data transfer in deparser.

4.4 Summary

We addressed three problems in this chapter — automatic generation of parser, automatic generation of deparser and possible way for scaling of deparser to higher throughput. The automatic generation of parsers uses the HFE M2 architecture which was introduced in [58] (see Sec. 4.1 for more details). We introduced an effective and general algorithm for mapping from P4 language to parser's architecture. The Sec. 4.3.1 introduces experiment results for generated and hand-written parsers. We can say that generated parsers are approximately two times worse than hand-written parsers in terms of latency and consumed resources.

The automatic generation of deparser from P4 language was introduced in Sec. 4.2. We introduced an effective and general algorithm for mapping from P4 language to novel deparser's architecture. The Sec. 4.3.2 introduces experiment results for generated deparsers which are capable to reach throughput of 100 Gbps. However, there are some packet lengths where the insertion process wastes available data bus. The text also introduces a possible way for scaling of deparser to higher throughput.

The transformation from P4 to HDL code of parser/deparser, including the detailed description of both architectures, was published in [A.1, A.2, A.12, A.8, A.10].

Chapter 5

Match+Action Processing Pipeline

In this chapter, we introduce results of our research regarding the Match+Action processing pipeline with details about transformation algorithm and generated architecture. We introduce mapping from P4 language to Match+Action groups, Match+Action tables and Match+Action routers which are the main building blocks of our pipeline. We also provide more details about usage of HLS which can be beneficially used for easy extension of available actions.

5.1 Main Building Blocks

The main building block of our architecture is the *Match+Action group* which was briefly introduced in Sec. 3.2.3. We also introduced the idea for mapping of P4 language constructs to Match+Action pipeline elements in Sec. 3.3. As we notice before, each *Match+Action group* represents one *control program* from translated P4 source and it contains one or more pipeline elements (tables or routers). Such group of building blocks is used for implementation of complex processing of parsed data. The following text introduces detailed architecture of proposed blocks.

We noticed that elementary building block is the *Match+Action table* which implements functionality of protocol field processing based on user defined P4 program. The engine performs three operations: (1) searching for the most suitable operation regarding to used headers and metadata, (2) computation of next used Match+Action element address and (3) execution of selected action on headers and metadata. During our research, we identified two types of pipeline engines:

1. *Match+Action table*
2. *Match+Action router*

The first type, full *Match+Action table*, is the engine with full functionality which was described in previous text. It contains engines for searching of the most suitable action (i.e., the match engine), engines for computation of next table address and engines for

execution of selected action on incoming headers and metadata. We provide more details about this Match+Action block in Sec. 5.1.3.

The second type, *Match+Action router*, is the engine with limited functionality. It just computes the next table or router address from incoming headers and metadata values. Such node allows us to implement (or represent) the conditional selection of next table or router address without instantiation of search and action engines. In other words, it allows us to map the *if-else* statement more directly from P4 source to Match+Action processing pipeline. We provide more details about this engine in Sec. 5.1.2.

The architecture of the Match+Action processing pipeline is published in this dissertation thesis because it is the latest result of our research. We start the introduction of our approach with description of mapping from P4's *control program* to pipeline of Match+Action elements.

5.1.1 Control Program

We briefly introduced the structure of *control program* in Sec. 3.3.3. It indirectly defines a sequence of Match+Action elements (tables and routers) and we can see it as an imperative program. The program may apply tables, call other control flow programs or test conditions. The execution of a table is invoked by the *apply* statement and it has four modes of operation (more information in [2]): (1) sequential, (2) selection based on executed action, (3) hit/miss check and (4) conditional selection based on *if-else* statement. The first mode, sequential, is the unconditional movement to the next table or router in control flow. The second mode is the conditional selection based on executed action in current table. The third mode performs selection of next pipeline element based on hit/miss event in actually used table. The test condition is classic *if-else* statement which is similar to C language conventions. Examples of each mode are given below this text.

```
control Ingress {
    // Sequential mode
    apply(table1);
    apply(table2);

    // Action selection mode
    apply(table3) {
        action1 : {apply(table4)};
        action2 : {apply(table5)};
        default : {apply(table6)};
    }

    // Hit/miss check
    apply(table7) {
        hit     : {table8);}
        miss    : {table9};}
    }
```

```
    // Test condition
    if(valid(ipv4) and ipv4.ttl < 42){
        apply(table10);
    }
}
```

The above example introduces the program called *Ingress*, including ten tables. The fist three tables are used in sequential manner. Our program follows by invocation of table regarding to executed action in *table3*. The example also introduces the usage of default rule for coverage of situation without any matched action. The control flow of our program continues with invocation of *table7*. Next tables, *table8* and *table9*, are invoked based on hit/miss result in *table7*. Finally, our example introduces a simple test condition for IPv4 header. The header is checked for validity and value of *ttl* field. The *table10* is used if both parts of the condition are satisfied.

As you can see from provided example, the *control program* describes the structure of the *Match+Action group* in terms of used Match+Action elements (i.e., used tables and routers). Therefore, our main task is to use the description of *control program* and generate the structure of Match+Action pipeline. We introduce details of this transformation in next section of this text.

5.1.1.1 Generation of the Pipeline Structure from Control Program

The main idea of the transformation is similar to identification of parser's structure which was introduced in Sec. 4.1.1. We described a *Parser Graph Representation* (PGR) as an input to our transformation algorithm. The PGR is an oriented acyclic graph which represents relations between protocols. After that, we used the *depth-first-search* algorithm for identification of Protocol Analyzer position in generated parser pipeline. The proposed idea can be reused for identification of the Match+Action pipeline structure. However, we cannot reuse the graph because all nodes of PGR have the same type (i.e., each node represents the Protocol Analyzer block). The *Match+Action group* contains two types of blocks: *Match+Action tables* and *Match+Action routers*.

The new tree structure reflects the previous need. Each node of refined graph represents one Match+Action table or router and each transition represents the next used Match+Action element. Condition of each transition is inferred from P4's *control program*. The refined structure also defines a common root node which is used as a starting point for processing of different *control program*s using one algorithm. All remaining properties of the refined structure stay the same (i.e., the graph is oriented and acyclic). This structure is built using the *depth-first-search* (DFS). The example of refined tree structure is shown in Fig. 5.1.

Generation of the Match+Action pipeline uses the refined tree structure as an input. The main problem is the same as generation of parser's structure from PGR representation. Therefore, we reuse the algorithm for identification of the longest path to each node in

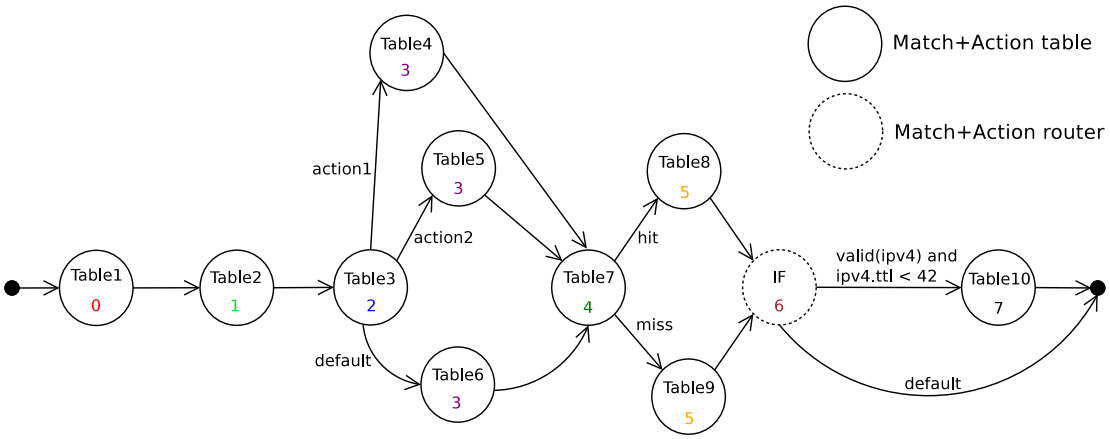


Figure 5.1: Graph of the control program from the example.

refined graph. That is, we use the Alg. 1 from Sec. 4.1.2. The result of the algorithm is shown in Fig. 5.1 and the longest path to each node is expressed by a number. Finally, all nodes are connected in any non-decreasing order. The example of such pipeline is shown in Fig. 5.2. The Match+Action engine does nothing if it is not selected (i.e., it passes input to output without any change). The structure of Match+Action pipeline follows architectures of parser and deparser. That is, each engine can be followed by optional pipeline for tuning of final frequency or latency. We introduce the architecture and transformation from P4 to Match+Action pipeline in following text.

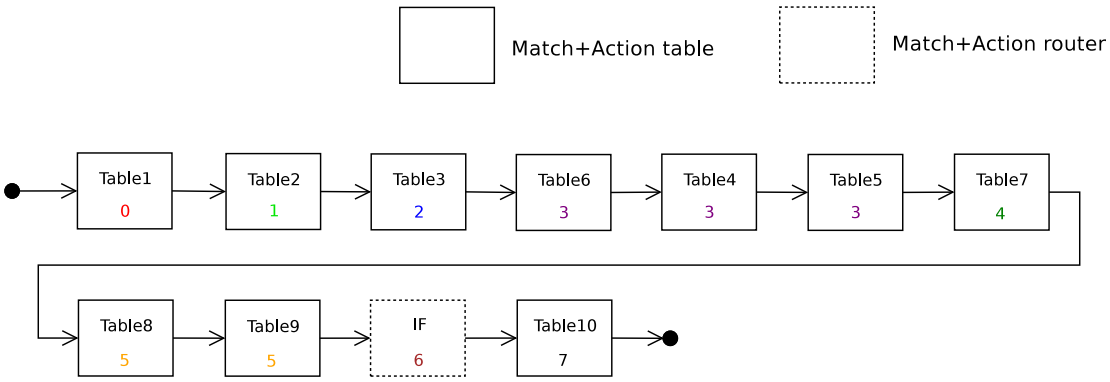


Figure 5.2: Generated Match+Action pipeline from the example.

5.1.2 Match+Action Routers

The *Match+Action router* is the second engine which is used in our architecture. As we noticed before, it is engine with limited functionality because it is used for implementation of *if-else* conditions. The engine computes the next table or router address from incoming header and metadata fields. Therefore, the engine does not need to implement any match or action sub-engine.

The *if-else* condition can contain field or header references, valid operator (used for detection of header validity), binary operators (including arithmetic and logic operations), unary and relation operators (e.g., less than, equal, less or equal, and so on). Therefore, our engine has to support a variety of common operations for selection of the next table or router address in pipeline.

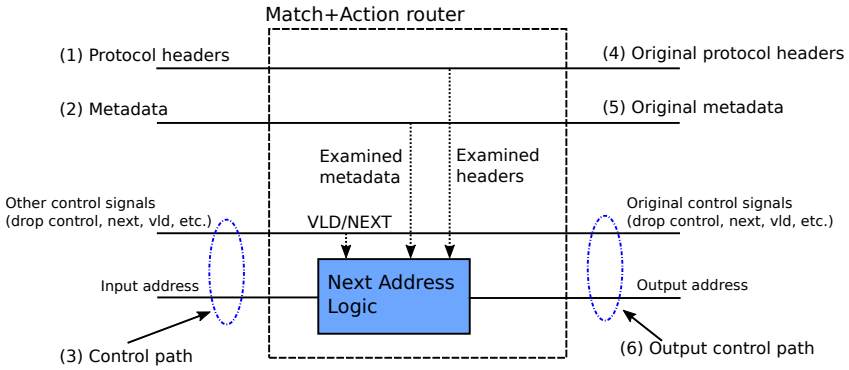


Figure 5.3: Architecture of Match+Action router.

The architecture of Match+Action router is shown in Fig. 5.3. The proposed interface is common for both engines (router and table) and it is also used for easy connection between engines. This interface provides the input information necessary to process one single condition. That is: (1) protocol headers, (2) incoming metadata and (3) control path signals (including table or router address, drop control, next/valid signalization, and so on). The output information includes (4) original protocol headers, (5) original metadata and (6) output control path. The last mentioned output is modified by the router engine because the control path contains the address of next element (table or router).

Each Match+Action router contains *Next Address Logic* block. The block is generated from scratch and it is configured with address of the router (i.e., it is configured with address in the pipeline). This information is used for activation of the engine which leads to computation of the next element address. Any headers, metadata or control signals aren't modified if address differs from the configured one. The behavior of *Next Address Logic* block depends on condition in the *if-else* statement. The translation of this condition is based on *Expression Tree* which is translated into HDL language (VHDL in our case) by *in-order* tree traversal. An example of this process is shown in Fig. 5.4. The example also shows the parallel nature of HDL language because all lines of provided code are executed at the same time.

The example introduces the process of transformation for the condition from previous section. Firstly, each condition is transformed to the *Expression Tree* where each node represents an operation (arithmetic, logic or relation) and each leaf represents a value, referenced header or field. Secondly, we use the *in-order* tree traversal for generation of the VHDL representation. The generated code reflects proposed functionality needs. It selects the next table address if generated condition was satisfied and engine was selected (i.e., the IN_TABLE_ADDRESS value equals to the address which was assigned to the

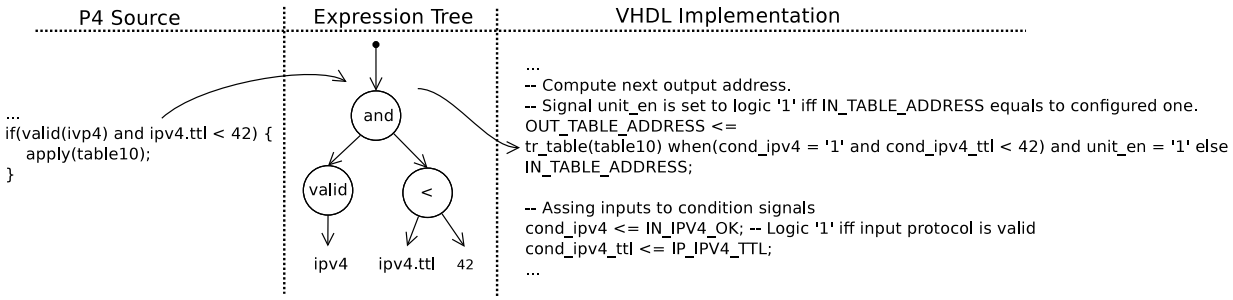


Figure 5.4: The example of transformation from P4 to the *Next Address Logic* block.

engine during translation of P4 source). The input address is transferred to the output unchanged if the engine wasn't selected.

5.1.3 Math+Action Tables

The *Match+Action table* is important engine of Match+Action pipeline because it implements match and action engines for further processing of header and metadata fields. The engine performs three operations: (1) searching for the most suitable action, (2) computation of next used Match+Action router or table address and (3) invocation of selected action on header and metadata fields.

Behavior of Match+Action table engine is inferred from the following P4 language constructs: Match+Action Table definition, Action definition and Protocol Header definition. The first language construct is used for specification of required match fields (including matching algorithms) and supported actions in this engine. The structure of incoming header and metadata fields is described by Protocol Header definition. Finally, the behavior of supported actions is described by Action definitions. There are two types of actions — user defined and primitive. The user defined actions have a list of required parameters and unique name across the P4 program. Each user defined action contains calls of primitive or other user defined actions. Therefore, we can implement quite complex operations which can be executed on incoming header and metadata fields. The list of primitive actions includes modification of header fields, insertion or deletion of protocols from packet, arithmetic operations on incoming data, and so on. The behavior and list of primitive actions is defined in P4 language specification [2].

The architecture of Match+Action table engine is shown in Fig. 5.5. Each pipeline engine uses the interface from Sec. 5.1.2. This interface provides the input information necessary to process a table request. That is: (1) input protocol headers, (2) input metadata,(3) control path signals (including table or router address, drop control, and so on) and (4) configuration interface which is used for filling of rules with corresponding actions and parameters. The output interface includes (5) protocol headers, (6) metadata and (7) control path signals. The output headers are changed based on selected action. All input protocol headers and metadata fields are stored in auxiliary FIFO memories. These memories are used to cover the initial latency of *Search engine*. Examined protocol

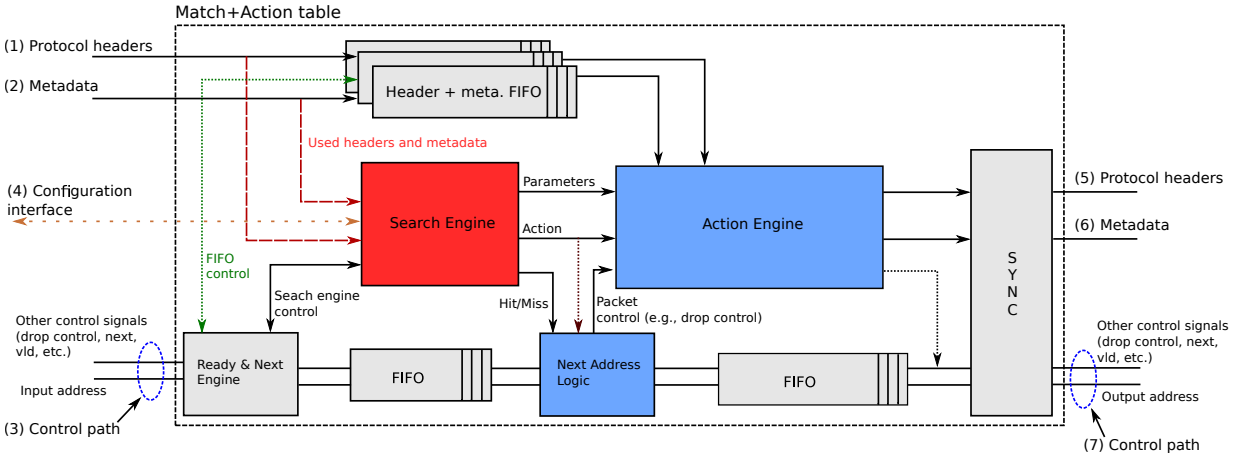


Figure 5.5: Architecture of Match+Action table.

headers and metadata fields are passed to *Search engine* which is responsible for searching for the most suitable action. The set of used protocol header and metadata fields is inferred from P4's Match+Action Table declaration (*reads* statement). The output of the engine provides information necessary for processing in *Action engine* (i.e., selected action and parameters) and computation of the next table or router address (based on executed action and hit/miss information). All input blocks (FIFO memories and *Search engine*) are controlled by the *Ready&Next* engine. The engine is configured with address of the Match+Action table during translation of P4 source code. The configured address is used for activation of Match+Action table if the input address equals to the expected one. The *Action engine* accepts packet control information (drop contol for example), protocol headers and metadata (from FIFO memories), and selected action (including parameters). This engine is generated from scratch for each *Match+Action table* in design. The behavior of user actions is inferred from P4's Action definitions. We describe the *Action engine* in Sec. 5.1.4. Finally, the last output block (SYNC) is used for synchronization of results from *Action engine* and FIFO to output.

We identified three types of blocks from proposed architecture: (1) Configured (grey color), (2) Fully table specific (blue color), and (3) Mixed (red color). The first group is general enough for usage in all *Match+Action tables*, only with different parameter settings like data width, assigned address in pipeline, and so on. The second group of blocks, table specific, are fully generated regarding to P4's Match+Action Table declaration. In our case, we fully generate the *Next Address Logic* and *Action engine*. The *Action engine* and its generation was briefly described in previous text and we provide more details in Sec. 5.1.4. Finally, the third type of blocks is the example of mixed architecture where some parts are fully generated and some parts are configured by parameters during translation.

Generation of *Next Address Logic* is quite straightforward because P4 language defines three modes for selection of next address (action based, sequential and hit/miss check). Therefore, the generation of the code is easier because we don't need to deal with complex structures of test conditions (*if-else* statements). The logic of HDL code generation stays

similar for different implementations of *Match+Action table*. Main differences are in selection of table or router addresses based on executed action or current hit/miss result in *Search engine*. The example of transformation, including the structure of generated VHDL code, is provided in Fig. 5.6.

The *Search engine* is used for mapping of incoming protocol and metadata header fields to used actions and parameters. The structure of this block can be complex because P4 language allows usage of different match algorithms for each used protocol or metadata field. We introduce the architecture of this block in following section.

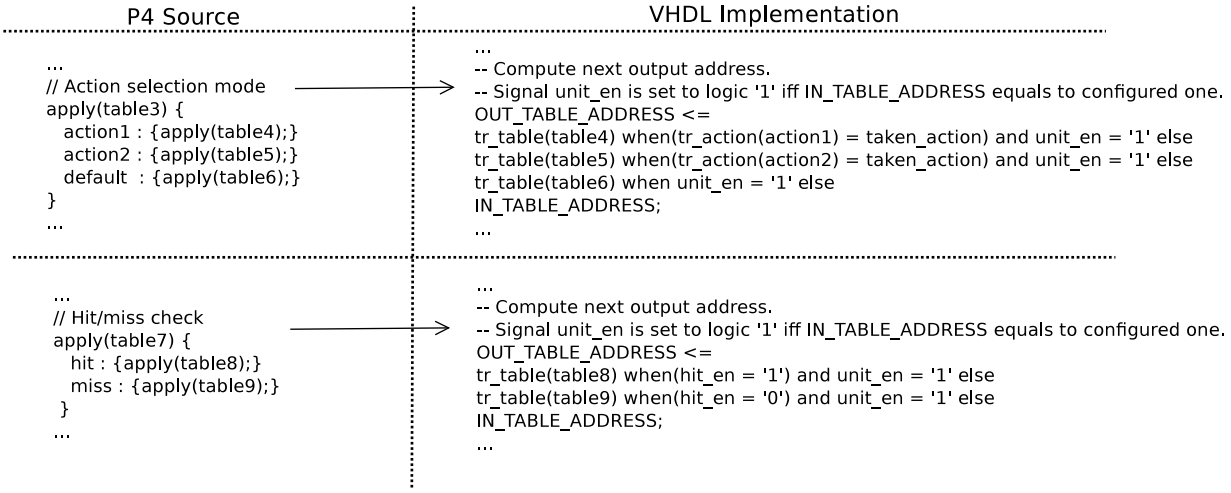
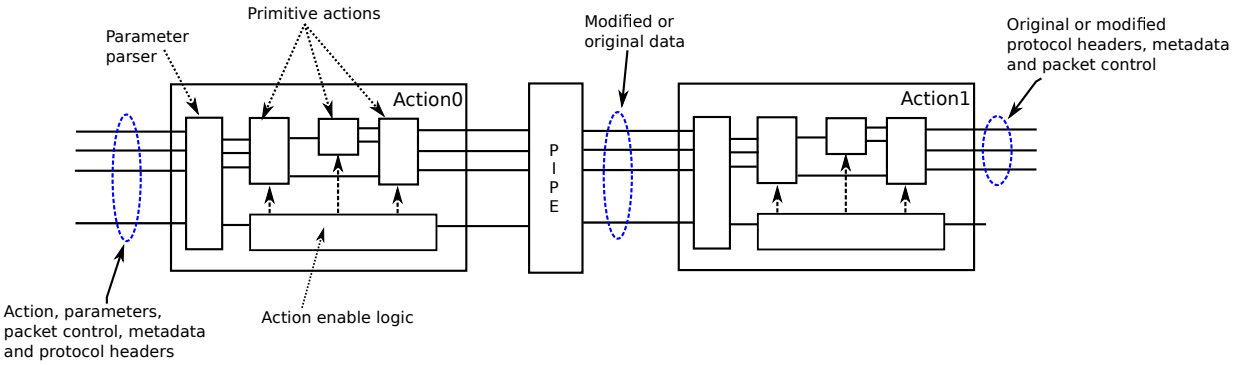


Figure 5.6: The example of transformation from P4 to the *Next Address Logic* block.

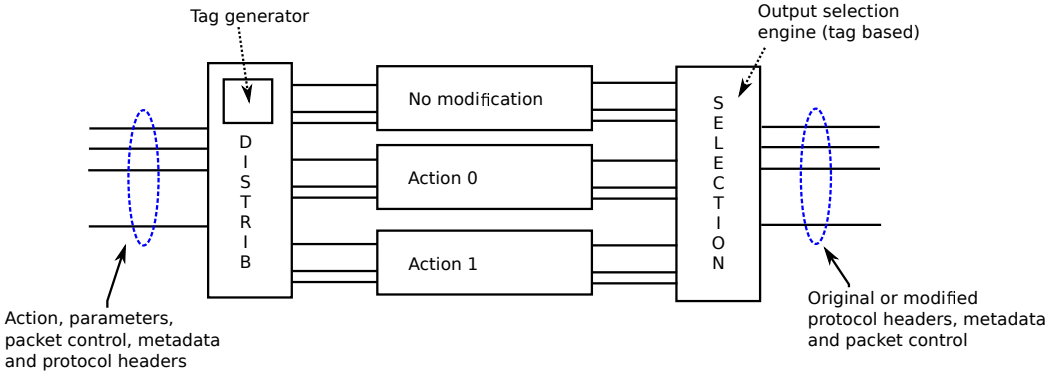
5.1.3.1 Search engine

The P4's Table specification defines a list of used protocol header fields together with matching algorithm. Current version of the language supports following algorithms: range, ternary, Longest Prefix Match (LPM), exact and header validity checking.

Design of general classification approach (suitable for P4) is out of scope of this thesis. Therefore, we utilize a TCAM because it implements exact, ternary and range match (after some changes [59]). Each row of TCAM contains one rule. In this approach, all used protocol headers and metadata are concatenated to create one wide lookup key to TCAM. Returned vector contains information about all matched rows (in one-hot encoding). The vector is passed to the priority encoder which identifies the address of most suitable rule. The computed address is used as an input to rule memory which contains data with encoded action and parameters. We also identify the common interface for *Search engine* which is used for communication with neighboring blocks (see Fig. 5.5). That is: (1) used protocol headers, (2) used metadata, (3) search engine control (VLD/NEXT signaling) and (4) interface for configuration from software. The output includes information necessary for computation of the next table address (based on hit/miss check or executed action). The *Search engine* is also equipped with a special configuration bus which is used for filling



(a) Serial architecture.



(b) Parallel architecture.

Figure 5.7: Basic architectures of Action engine.

of rules with corresponding action and parameters. Finally,the series of outputs are used for passing of selected actions and parameters to *Action engine*. This interface allows easy and fast replacement with more advanced *Search engines*.

5.1.4 Action Engine

The *Action engine* performs operations on packet control information (drop control for example) and incoming metadata or protocol header fields. The outputs of this block are modified protocol or metadata fields based on selected action from *Search engine*. Supported table actions are provided in P4's Match+Action Table definition (*actions* statement; more details in Sec. 2.3.5). Each record of *actions* statement is a user defined action with a unique name in P4 program. In general, each user defined action is a compound action consisting of primitive and other user defined actions. The specification and set of supported primitive actions is defined in P4 language specification [2]. Therefore, each user defined action can be expressed using a list of primitive actions which are connected together. Consequently, we can define the library of all primitive action in HDL and connect together all used actions. There are two architectural approaches for arrangement of action engines: *serial* or *parallel*. Brief descriptions of both architectures are provided in Fig. 5.7.

The first one, serial, is the example of classic architecture for implementation of processing pipeline. This approach is shown in Fig. 5.7a. It contains two sub-engines — *action blocks* and *pipelines*. The *pipeline* is an optional block which is used for tuning of final latency and consumed resources. The most important part is the *Action block* which implements the functionality of processing pipeline. Each *Action block* has similar structure and it can be seen from the proposed figure. Firstly, action parameters are parsed in *Parameter parser* for easier usage of action parameters in *Action blocks*. Finally, the *Action enable logic* is used for controlling of reaction on incoming action value. The engine is enabled if the value of action equals to configured value which is assigned to action during translation of the table's P4 source. Incoming data are transferred through the action block untouched if the value of action differs from the expected one. The advantage of this approach is a regular structure which is suitable for generation from abstract description. Unfortunately, this architecture is not suitable for latency sensitive applications because data pass through all action blocks. Therefore, the total latency can be expressed as the sum of all latencies on the path. In our case, higher latency leads to higher memory requirements on Payload Buffers.

The second architecture, parallel, is a different approach for arrangement of action blocks. This approach is shown in Fig. 5.7b. The input block of this architecture is the *Distributor*. This block is used for distribution of incoming data to action blocks based on the value of action from *Search engine*. The *Distributor* block also contains engine for assignment of synchronization tags. The synchronization tag is used for selection of output data from action blocks because we need to preserve the order of requests during data processing. This approach is suitable for shortening of latency because a request doesn't need to be passed through all action blocks. The latency for processing of input data depends on actual distribution and usage of action blocks because distributed requests are synchronized by *Selection* block.

From the proposed description of *Action engine* architectures, we rather use the parallel architecture because the processing latency depends on distribution of requests. The P4 language defines a fixed set of primitive actions. Unfortunately, this limitation is not suitable for future development because we typically need to extend the current action set with new actions (e.g., computation of advanced statistics, pattern matching, and so on). The user defined actions can be described using the HLS which is beneficial for fast development. Moreover, the source code of new action can by provided by administrator, mathematician or security expert. The next text provides details about usage of HLS in high-speed environment. The results of our research were verified in SDM [29] project which is actively used for monitoring and protection of computer networks. We also demonstrated our research in several network applications (NetFlow and non-NetFlow kinds of data processing) which were published in [A.4, A.5, A.6, A.9].

5.2 Extending P4 with Non-Standard Actions

The parallel architecture of Action engine is similar to *SDM Update* block which is an application specific network processor for a stateful flow-based network traffic measurement. We introduced this approach in [30]. The proposed work also introduces usage of HLS in high-speed network environment which helps us with faster development of new processing engines. This approach can be also used for extensions beyond P4 language because the set of primitive actions is limited. There are situations when we need to support more complex operations which cannot be implemented with actual set of primitive actions. There are two possibilities for implementation of new action. The first possibility is to use the HDL language which is the most common approach because the architecture and implementation details are under control. As a result, we can get fast and efficient implementation of new action. However, the development time can be quite high compared the utilization of HLS. Rather, we can describe new action in C/C++ language and connect it to the existing infrastructure. The results of this research was also used by Kekely et al. in [60]. We demonstrate our approach on *SDM Update* in next section. The main proposed ideas can be also used in the case of *Action engine* for implementation of more complex instructions in shorter time.

Software Defined Monitoring (SDM) [29] relies on advanced monitoring tasks implemented in software which is supported by a configurable hardware accelerator. The monitoring tasks reside in the software and can easily control the level of detail retained by the hardware for each flow. This way, the measurement of bulk/uninteresting traffic is offloaded to the hardware, while the interesting traffic is processed in the software. SDM enables creation of flexible monitoring systems capable of deep packet inspection at high throughput.

5.2.1 SDM Architecture

SDM consists of two main parts (firmware and software) connected together via PCI Express bus. Both parts are tightly coupled together to allow a precise control of the traffic processing on the per-flow basis. The software part of SDM consists of the controller and monitoring applications. The advanced monitoring tasks, such as analysis of application protocols, are performed in the monitoring applications. The management of the hardware (removing and insertion of the processing rules) is performed in the controller. The SDM Processor hardware is controlled by instructions stored in the rules. The instruction tells the hardware what action must be performed for each input packet. The hardware passes the data to the software in the form of packet metadata (i.e., extracted information from the packet header) or aggregated flow records (NetFlow for example). Whole received packet can be also sent to software for deeper analysis. Graphical representation of the SDM concept is shown in Fig. 5.8. The following text is focused on description of SDM Processor.

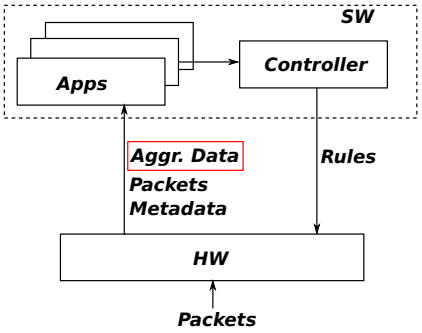


Figure 5.8: Brief SDM Architecture.

5.2.2 SDM Processor Description

For the scope of this work, the hardware part of SDM is the most interesting. SDM Processor is conceptually an application specific network processor for a stateful flow-based network traffic measurement. The main parts of the SDM Processor are shown in Fig. 5.9. The processing of all incoming packets starts with header parsing and extraction of interesting metadata in the *Parser* block. This prepared data is passed to the *Search* engine which performs lookup in the table of rules (these rules are maintained by the software SDM Controller). If the rule is found, it is sent with the data to the *SDM Update* engine which performs the operation with respect to the passed instruction. If the rule is not found, default behavior is performed (packet is typically passed to the software for further analysis).

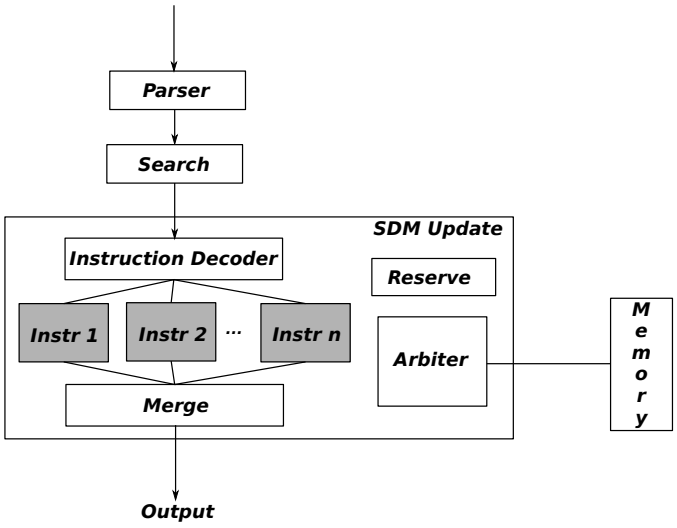


Figure 5.9: Brief SDM Update architecture and its connection in SDM concept.

Update

The *SDM Update* engine brings the statefulness to the SDM Processor. It manages the flow records — updates the aggregated data. It mainly updates the stored values based on the input data and the rule action. The action for every packet has the address of the record and a specification of the operation (aggregation type). Update of the record is delimited by two memory operations: read the stored values of the record and write back the updated values. Special type of operation is the export of record values, possibly followed by clearing the record values in the memory.

Infrastructure around *Instr x* blocks is designed with respect to easy implementation of new instructions. Blocks of this generalized environment are shown in Fig. 5.9. The main services of this infrastructure are:

- Decoding and forwarding of a request to the appropriate processing engine.
- Memory record reservation (to avoid RAW data hazards).
- Memory access arbitration.
- Output synchronization (to preserve the ordering of requests).

The idea for fast implementation of new instructions is following: As a new network monitoring software plugin is installed to SDM, it is accompanied by a description of the most time consuming task in C language. The task is then synthesized to VHDL and inserted directly to the SDM hardware in the form of a new instruction block. The FPGA firmware has to be resynthesized in our system, but the partial dynamic reconfiguration could allow dynamic changes of the SDM Processor instruction set in the future. During the runtime, the SDM controller software installs the processing rules to the hardware. The hardware then executes the new instruction for each packet of selected flows.

To insert the new instruction engine to the *SDM Update*, the surrounding sub-engines around the instruction blocks stay unchanged. Hence, we implement instruction engines in HLL (C or C++ in our case), perform HLS of this code, connect the generated instruction engines and add routing information into the *Instruction Decoder* routing table. This process of development consumes less time and allows faster implementation of the new hardware accelerated function. Moreover, final solution is verified during implementation of new accelerated block because modern tools for HLS require implementation of the testbench file. After this verification, new engine is easily connected into the existing and tested infrastructure. As a demonstration, we implement following instructions:

- The **Identity (I0)** instruction is used for transformation of input request to the predefined output format. This engine doesn't perform any update at all, it is rather used for data output synchronization and transformation. The Identity engine is implemented in VHDL language because it is considered to be the part of the SDM Processor infrastructure.

- The **NetFlow (I1)** instruction is used for NetFlow aggregation. It updates the flow start/end timestamp, increases packet/byte counters, and performs logical *or* of TCP flags. NetFlow engine is implemented in both VHDL and C languages for comparison.
- The **NetFlow Extended (I2)** instruction demonstrates an easy extension of existing C implementation. Basic update functionality of this instruction is the same as described in I1. In addition, I2 stores the TCP flags of the first five packets. This additional information may become very useful for analysis of TCP handshake or for detection of network attacks like DoS. The NetFlow Extended engine is implemented in C language with usage of the existing code from I1.
- The **TCP Flag Counters (I3)** instruction is the implementation of non-NetFlow update. The instruction performs incrementation of counters of observed TCP flags. For example, one can see the number of ACK flags transmitted during the whole TCP connection. Information from this aggregate can be used to support advanced flow analysis [61]. This engine is implemented in C++ language.
- The **Timestamp Diff (I4)** instruction is another implementation of non-NetFlow update. This instruction creates an aggregated record of inter-arrival times of the first eleven flow packets. These times are computed with nanosecond precision. The engine is implemented in C language. Information from this aggregated record can be used as a network discriminator for flow-based classification [61] or for identification of application protocol [62].

5.2.3 Results

To verify the designed *SDM Update* part of the SDM Processor, we have implemented a prototype of the described architecture. We choose the board equipped with Virtex-7 H580T FPGA which is powerful enough for processing of 100 Gbps traffic [6]. The Xilinx Vivado HLS version 2013.2 was used for HLS (C to VHDL). The Xilinx ISE version 14.6 was used for VHDL to FPGA netlist synthesis. We have disabled all synthesis optimizations to avoid the register duplication and other modifications that can lead to higher frequency but also to higher resource usage.

The resource usage of the instruction blocks is shown in Tab. 5.1. We compare the resources of hand-written VHDL code to the HLS based implementation of the I1 instruction. One can see that the hand-written implementation occupies less FPGA resources and the final frequency is higher than HLS based solution. However, the time needed for VHDL implementation is approximately two times longer than for the solution synthesized from C language. The HLS is also very useful for non-HDL programmers because they can create hardware accelerated engine very easily from the C/C++ source code.

Tab. 5.2 shows the total resource utilization of the whole *SDM Update* engine. Each line of this table lists the instructions supported in synthesized engine. Frequency results from HLS are good enough for processing of the high-speed network traffic. We consider

the minimal frequency of 200 MHz and the initialization interval of one as the requirement for 100 Gbps traffic processing. All synthesized variants of the *SDM Update* meet the frequency requirement for processing of 100 Gbps traffic.

Instruction	**Slice Reg** [-]	**Slice LUT** [-]	**Frequency** [MHz]
(I0)Identity	495	504	627.559
(I1)NetFlow (handmade)	1754	325	425.134
(I1)NetFlow	1846	824	308.641
(I2)NetFlow Extended	2070	1113	308.641
(I3)TCP Flag Counters	0	1046	327.868
(I4)Timestamp Diff	5199	2556	306.748

Table 5.1: Resources of instruction blocks.

Instructions	**Slice Reg** [-]	**Slice LUT** [-]	**BRAM** [-]	**Frequency** [MHz]
I0	478	1345	0	392.057
I0,I1	3453	5024	24	294.772
I0,I2	4867	4399	24	281.897
I0,I3	3012	4636	24	296.002
I0,I4	7074	6598	24	286.090
I0,I1,I3	5704	8107	48	285.531
I0,I1,I4	9762	10360	48	275.820
I0,I1,I3,I4	11965	12998	72	255.660
I0,I1,I2,I3,I4	16018	15969	96	238.792

Table 5.2: Resources of SDM Update.

5.3 Summary

We introduced and described our architecture and mapping from P4 language to general hardware representation of Match+Action processing pipeline. We identified two engines of our architecture: *Match+Action router* and *Match+Action table*. The first engine, *Match+Action router*, is used for computation of next table or router address based on prescription of *if-else* statement. The second engine, *Match+Action table*, is used for implementation of P4's match and action functionality. Both engines are generated based on the description of translated P4 source. We also identify the common interfaces for all proposed blocks of Match+Action pipeline. This step is very important for future research because one can easily swap the old implementation of a block with a new one. This allows us to drive the research in terms of implementation/verification of new ideas and

architectures. We also introduced two architectures of *Action engines* with pros and cons of each approach.

After that, we introduced the usage of HLS in *SDM Update* block which has similar architecture that we use in the Match+Action processing pipeline for easy extension with new operations in shorter time. We demonstrate that parallel architecture and HLS based instructions are capable to process traffic at speeds of 100 Gbps and beyond. We also demonstrated and validated our research in several network applications (NetFlow and non-NetFlow kinds of data processing) which were published in [A.4, A.5, A.6, A.9].

Chapter 6

Use Case Study

The following section discusses results for three use cases of full P4 pipeline, including match and action functionality. We provide results of required resources, throughput and number of generated lines, together with required time for compilation. Using these three use cases, we demonstrate easy extensibility with new protocols and actions. Both aspects (extensible set of supported protocols and actions) are important for fast deployment of new applications in these days.

6.1 Test Use Cases

In this section, we provide three use cases which were selected for demonstration of flexibility and our architecture. Namely, we will work with following P4-based devices:

1. **IPv4 Filter** is an example of simple engine where the filtering process is based on source IPv4 address. The list of allowed IPv4 addresses and corresponding actions is uploaded at runtime. Two actions are possible: pass or drop the packet. The filter also drops all non-IPv4 traffic by default.

2. **IPv4+IPv6 Filter** extends the IPv4 Filter with further support of IPv6 protocol. All non-IPv4 or non-IPv6 traffic is dropped.

3. **Full Filter** extends the IPv4+IPv6 Filter with further support of tagging functionality (VLAN or MPLS). Four actions are possible: pass, drop, VLAN tagging or MPLS tagging. All non-IPv4 or non-IPv6 traffic is dropped.

The following text provides results of required resources, throughput and number of generated lines. We selected this views because it allows us to demonstrate the usability and performance of generated architecture in real high-speed network environment. The results of proposed use cases are published in [A.7, A.10].

6.2 Experiments

All introduced use cases were described and translated from P4 language to VHDL using our generator. The generated code was connected to the NetCOPE [7] which is a general platform for rapid development of network applications on the family of COMBO cards. The VHDL code was translated for COMBO-100G card [6] using the Xilinx Vivado 2016.2 design suite. The card is capable to process 100 Gbps and it is equipped with CFP2 transciever cage, PCI-Express generation 3, QDR memory and Xilinx Virtex-7 XCVH580T FPGA. All use cases were successfully translated with 512 bits wide data bus at frequency of 225.8 MHz. Reached results and further details are provided in following text.

6.2.1 Required Resources

In this section, we introduce resource utilization for individual use cases. The resource consumption can be divided into two parts — resources of Match+Action pipeline and resources of parser/deparser.

Resource consumption of Match+Action pipeline is constant because this part of our architecture doesn't have any configurable pipeline stages. The number of tables, routers, processed packets per one second and required resources after synthesis are shown in Tab. 6.1. Each Match+Action table was configured to support 32 records in TCAM which

Figure 6.1: IPv4 Filter - Pareto optimal solution optimized for throughput and resource utilization with corresponding latency.

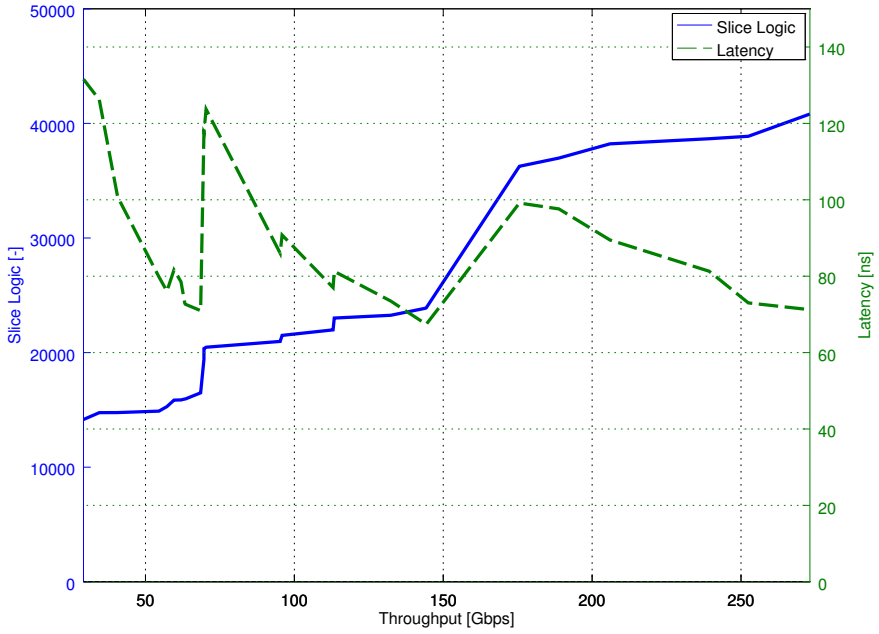


Figure 6.2: IPv4+IPv6 Filter - Pareto optimal solution optimized for throughput and resource utilization with corresponding latency.

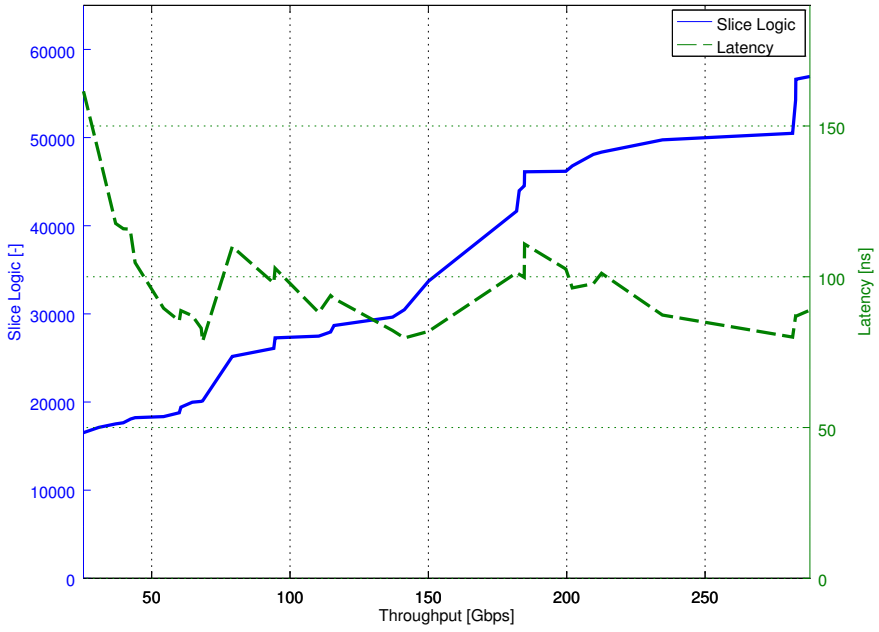


Figure 6.3: Full Filter - Pareto optimal solution optimized for throughput and resource utilization with corresponding latency.

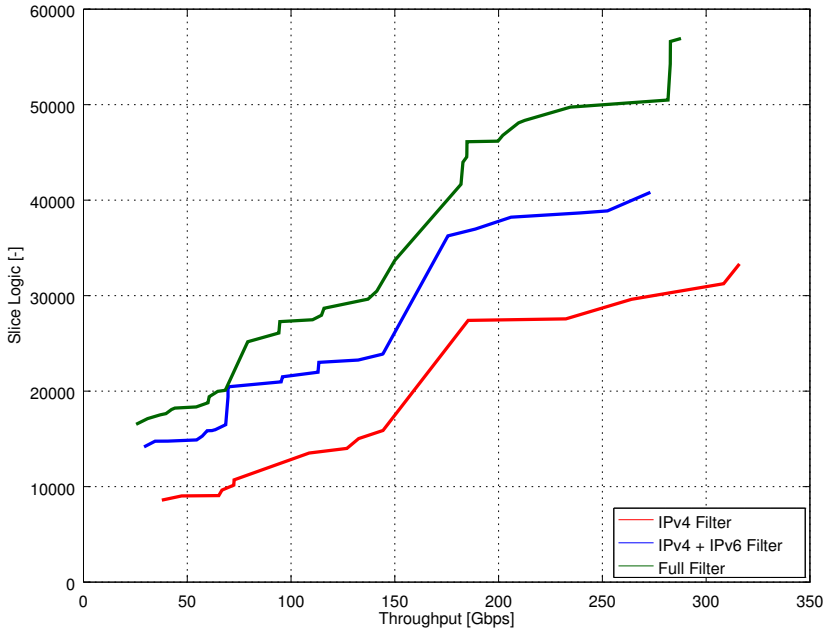


Figure 6.4: Comparison of the FPGA resource utilization versus throughput Pareto sets for tested use cases.

is enough for validation of proposed architecture. We provide required resources of *Search Engines*, including TCAM parts, in Tab. 6.2. The TCAM part can be replaced by more effective or novel classification approach which leads to different resources. All results are after synthesis for the Xilinx Virtex-7 XCVH580T FPGA using the Xilinx Vivado 2016.2 design tool.

The parser/deparser resource utilization is fluctuating because both blocks have configurable pipeline modules (i.e., each configuration has different requirements on Slice Logic). We developed an automatic testing environment which is capable to translate P4 source, to configure generated VHDL code (i.e., the tool setups pipeline stages in parser and deparser), to start the synthesis, and to extract results from the synthesis log. This allows us to investigate complex set of possible solutions. However, the set of all solutions can be quite large (for complex P4 program) and it can be time consuming to process all possible configurations of pipeline stages. Due to this, we extend the tool with possibility to select a limited set of configurations which allows us to briefly inspect properties of generated devices. The tool was configured to test three different input data widths: 256, 512 and 1024. These parameters, data width and configuration of pipeline modules, generate a large space of available solutions.

We provide several graphs: three showing Pareto sets optimized for throughput and chip area without any regard to latency, and one for comparison of different Pareto sets of individual use cases. Graph of Pareto set for given use cases contains two axes. The first

Project	Tables	Routers	Slice Logic [-]	Packet rate [Mpps]
IPv4 Filter	2	1	1917	335
IPv4+IPv6 Filter	3	2	5781	339
Full Filter	3	2	6709	318

Table 6.1: Required resources of Match+Action engines.

axis shows required chip area in term of Slice Logic (number of used LUTs plus FlipFlops). The second axis shows corresponding latency of those solutions (optimized for throughput and chip area). Results of individual use cases are shown in Fig. 6.1, Fig. 6.2 and Fig. 6.3. It is also important to notice that provided solutions aren't globally optimal because our data set doesn't contain all possible solutions. However, the approach with limited configuration set allows to briefly inspect properties of generated devices in a reasonable time. Finally, the Fig. 6.4 introduces comparison of Pareto sets optimized for throughput and chip area of provided use cases.

Project	Search Engines [-]	BRAM [-]	TCAM Engines [-]
IPv4 Filter	956	1	527
IPv4+IPv6 Filter	3529	1	2402
Full Filter	3907	1	2406

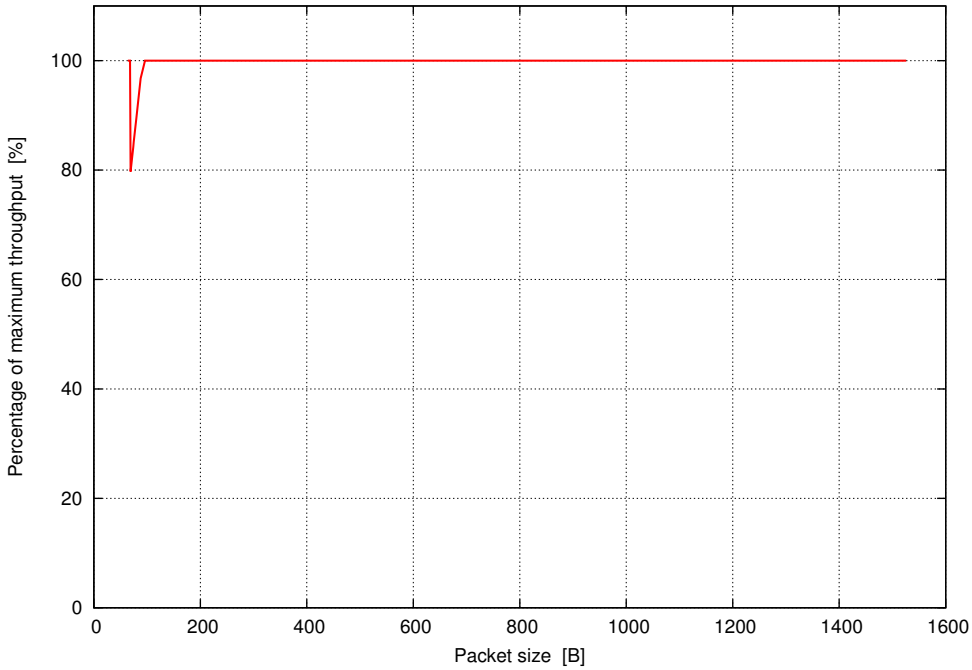
Table 6.2: Resources of Search and TCAM engines; Values are summarized through all tables in given project and they are expressed in term of Slice Logic.

6.2.2 Throughput

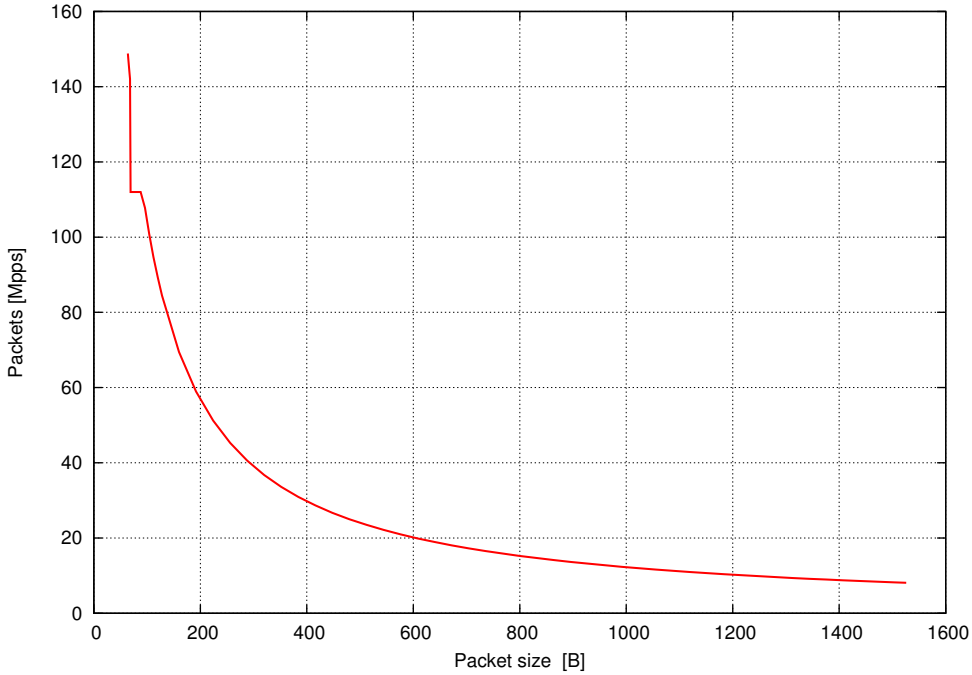
We measured throughput of all use cases in high-speed environment. We also created configuration tools capable to upload a user defined rule set. The rule set was prepared for utilization of each implemented functionality. Therefore, we simulate a situation when incoming traffic is distributed across all supported functions. The Fig. 6.5 shows percentage throughput of IPv4 and IPv4+IPv6 Filter for generated traffic at speed of 100 Gbps and a number of processed packets per one second. The Fig. 6.6 introduces the same but for the Full Filter use case.

All tests were performed against Spirent's Hypermetrics MX-100G-F1 module (SPT-N11U chassis). The proposed results show that our architecture is capable to hit 100 Gbps. However, there are some packet lengths when an inefficiency in deparser causes decrease of throughput. This inefficiency is related to insertion of protocol header before payload because each Protocol Appender works with aligned data at zero offset. Therefore, this optimization leads to less effective usage of data bus (i.e., 64 bytes wide data bus cannot be shared by more packets) which leads to decrease of output throughput.

The Full Filter has smaller throughput than IPv4 or IPv4+IPv6 Filter (compare short packet lengths in Fig. 6.7). This is caused by insertion of four additional bytes (MPLS

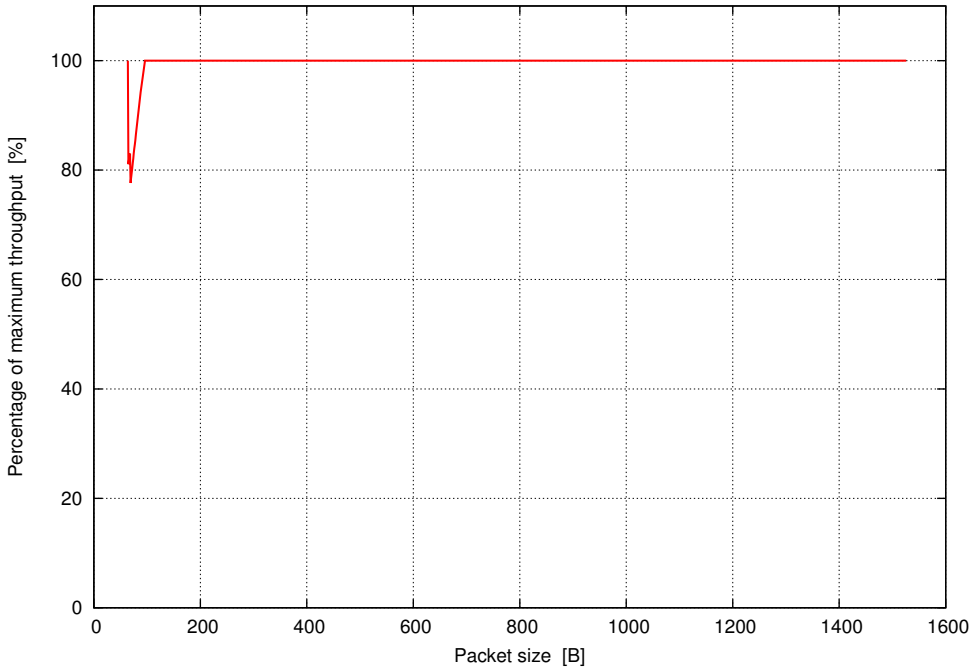


(a) Percentage of maximum throughput.

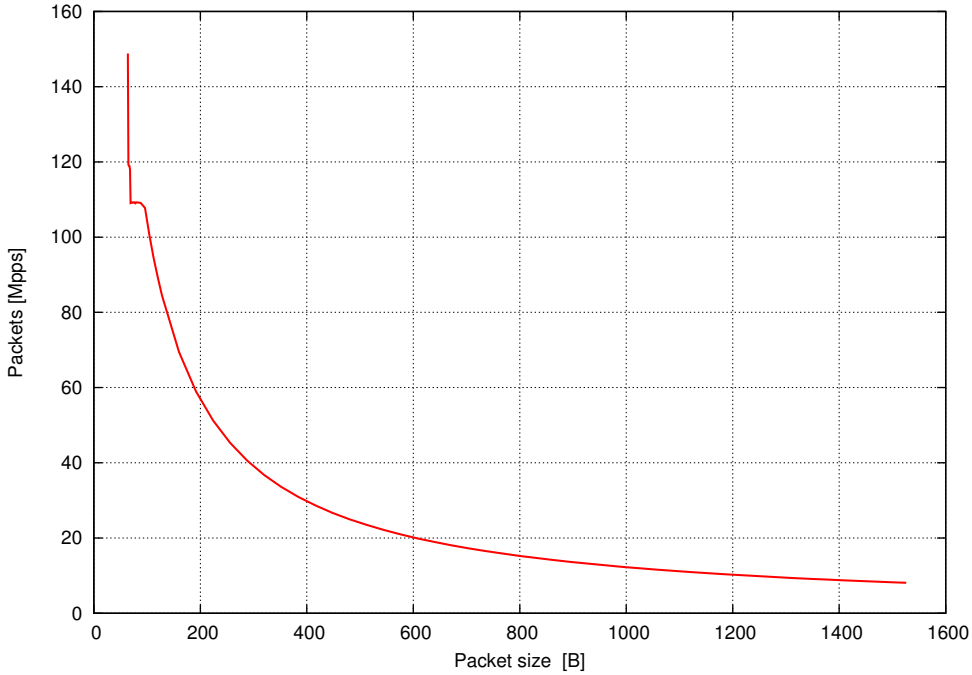


(b) Packet rate.

Figure 6.5: Throughput of generated IPv4 and IPv4+IPv6 Filter devices.



(a) Percentage of maximum throughput.



(b) Packet rate.

Figure 6.6: Throughput of generated Full Filter device.

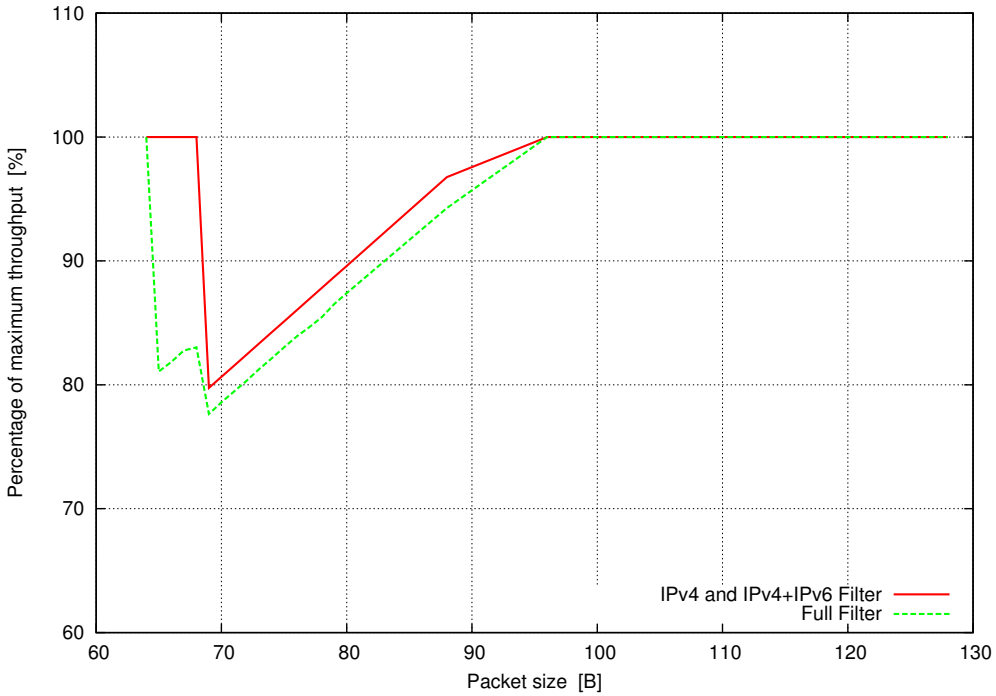


Figure 6.7: Detail of IPv4, IPv4+IPv6 and Full Filter throughput for small packet sizes.

or VLAN tag) to a packet which leads to decrease of input throughput because deparser pipeline needs to be stalled during insertion. The Full Filter tags approximately 62% of incoming traffic (31% with MPLS tags and 31% with VLAN tags).

6.2.3 Generated Code vs. Total Lines of Code

In this section, we introduce possible speedup in development of high-speed network devices. Reached results for individual use cases are shown in Tab. 6.3. The table contains compilation times of corresponding P4 source code (number of P4 source code lines is included). The **Generated lines** column expresses effort of the generator. For example, we generated 6283 lines of VHDL code from 91 lines of P4 source code. The **Total lines** is a sum of generated lines and lines of library source code (FIFO, TCAM, and so on). Table also shows that the generator produces approximately one third of each project. The VHDL code is generated from few lines of P4 source code during two seconds.

Project	P4 lines	Time [s]	Generated lines	Total lines
IPv4 Filter	91	1.574	6283	24791
IPv4+IPv6 Filter	129	1.818	9888	28396
Full Filter	212	1.929	13824	32332

Table 6.3: Generated code versus total lines of code.

6.3 Summary

This chapter provides results of our architecture which was generated from abstract description in P4 language. The text introduces three use cases — IPv4 Filter, IPv4+IPv6 Filter and Full Filter. These use cases demonstrate the flexibility and easy extensibility with support of new protocols and actions. All devices were generated in two seconds from several lines of P4 source. This makes the development more effective. For example, developer can be focused on implementation of novel hardware approaches related to classification because new module can be connected to the existing infrastructure. Text also provides Pareto sets optimized for throughput and resource requirements of individual use cases. Finally, all use cases were synthesized for COMBO-100G card and tested against Spirent network test device. Each use case is capable to hit throughput of 100 Gbps at frequency of 225.8 MHz on Xilinx Virtex-7 XCVH580T FPGA. The results from this chapter are published in [A.7, A.10].

Chapter 7

Conclusion

The aim of my dissertation thesis is to provide the process of mapping from abstract language to the architecture of network device which is suitable for automatic generation and capable to hit processing speed of 100 Gbps.

We propose the Parser-Deparser model. Our model of network device is based on the standard P4 model and it consists of three modules. The first module, parser, is used to break the incoming network data into individual header fields (i.e., it has the same functionality like parser in P4 model). The output of this module is a set of structured extracted fields, metadata and valid bits. All these inputs are passed to the processing block which does not strictly define the structure of processing part (compared to the standard P4 model where the structure of processing part is given). Therefore, we are able to connect strong ideas of P4 model and rapid implementation of novel network functionality (e.g., pattern matching, new architectures of network devices, novel classification and so on). Finally, the last module, deparser, is used for construction of network packet back from protocol headers and valid bits. This approach is general enough for variety of network applications. The Parser-Deparser model is capable to accommodate the P4 device model because the processing part allows mapping of Ingress Match+Action, Queues and/or Buffers and Egress Match+Action to predefined architecture.

We propose the architecture of our network pipeline which is suitable for automatic generation and capable to hit high-speeds. It is an example of the trade-off between complexity of high-speed network hardware blocks and transformation process. The proposed architecture contains several building blocks which are connected via predefined interfaces into deep high-speed pipeline. We introduced the mapping from higher level (P4 language) to the architecture of parser, deparser, Match+Action tables, Match+Action routers and Match+Action groups. The description on higher levels allows the user to be focused on implementation of network application without deep knowledge of underlying architecture. This approach was demonstrated on two examples — (1) transformation of P4 description to VHDL and (2) extension of available action set with usage of current HLS tool where the description of user action was provided in C/C++ language. Our results shows that both approaches are capable to produce synthesizable code of a network device which is

able to hit processing speed of 100 Gbps.

The transformation from the P4 to VHDL was demonstrated on three use cases — IPv4 Filter, IPv4+IPv6 Filter and Full Filter. These use cases show the flexibility and easy extensibility of our architecture with new protocols and actions. The behavior of each use case was described in P4 language and translated using our tool. We were able to reach the throughput in range from 77.6 to 100 Gbps and the frequency of 225.85 MHz on Xilinx Virtex 7 XCVH580T FPGA.

The transformation from C/C++ to VHDL was demonstrated on the extension of processing part with user defined actions. The P4 language defines a fixed set of primitive actions. Unfortunately, this limitation is not suitable for future development because we typically need to extend the current action set with new actions (e.g., computation of advanced statistics, and so on). The user defined actions can be described in higher language which is beneficial for fast development. Moreover, the source code of new action can by provided by administrator, mathematician or security expert. The results of this research were verified in SDM [29] project which is actively used for monitoring and protection of computer networks.

7.1 Contributions of the Thesis

The contributions of my dissertation thesis are summarized in the following list:

1. The modular architecture of network device which is suitable for generation of high-speed packet processing devices from the abstract description.

2. The structure and transformation process for individual parts of generated high-speed network device: parser, Match+Action router, Match+Action table and deparser.

3. The architecture of parallel Action Engine which is capable to process traffic at speed of 100 Gbps. The architecture of this engine was verified in SDM [29] project. It is also capable to accommodate High-Level Synthesized engines which are described in C/C++ language. Results of this research are used in [60].

4. The tool, which generates the proposed architecture of network device from the abstract description in P4 language.

5. Experimental results for parser, deparser and Match+Action processing pipeline. Finally, the text provides three use cases which were tested against Spirent's Hypermetrics MX-100G-F1 module (SPT-N11U chassis) at speed of 100 Gbps. Each use case was described in P4 language, translated to VHDL and tested in real hardware environment. The provided use cases demonstrate the flexibility and easy extensibility with support of new protocols and actions.

7.2 Future Work

The author of the dissertation thesis suggests to explore following:

- Provide a general architecture of *Check* engine.
- Investigate further optimizations of generated structure (e.g., automatic detection of similar protocols which leads to optimization of PGR's structure).
- Provide an effective architecture of deparser block, capable to construct packets at speeds above 100 Gbps.
- Provide architecture of more complex *Search Engine* in Match+Action table.
- Extend the Match+Action table with advanced stateful processing (inspired by OpenState [1]).
- Provide a study of this technology for speeds beyond 100 Gbps (e.g., for upcoming 400 Gbps Ethernet standard).

Bibliography

[1] OpenState SDN Project. OpenState Specification Version 1.0 (beta). Available from: https://github.com/OpenState-SDN/openstate-spec/releases

[2] The P4 Language Consortium. The P4 Language Specification. March 2015. Available from: http://p4.org/wp-content/uploads/2015/04/p4-latest.pdf

[3] Open Networking Foundation. OpenFlow. Available from: https://www.opennetworking.org/sdn-resources/openflow

[4] Facebook. Introducing "6-pack": the first open hardware modular switch. Available from: https://code.facebook.com/posts/717010588413497/introducing-6-pack-the-first-open-hardware-modular-switch/

[5] Facebook. Open networking advances with Wedge and FBOSS. Available from: https://code.facebook.com/posts/145488969140934/open-networking-advances-with-wedge-and-fboss/

[6] Liberouter. COMBO-100G. June 2016. Available from: https://www.liberouter.org/combo-100g/

[7] Liberouter. NetCOPE. Available from: https://www.liberouter.org/technologies/netcope/

[8] Postel, J. RFC 791: Internet Protocol. September 1981. Available from: http://www.ietf.org/rfc/rfc791.txt

[9] Deering, S.; Hinden, R. *RFC 2460 Internet Protocol, Version 6 (IPv6) Specification*. Internet Engineering Task Force, December 1998. Available from: http://tools.ietf.org/html/rfc2460

[10] Postel, J. RFC 793: Transmission Control Protocol. September 1981. Available from: http://www.ietf.org/rfc/rfc793.txt

[11] M. Mahalingam et al. RFC 7348: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. August 2014. Available from: http://www.ietf.org/rfc/rfc7348.txt

[12] IEEE Computer Society. *IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks*. IEEE Standard, IEEE, 2003, ISBN 9780738136622.

[13] Braun, F.; Lockwood, J.; Waldvogel, M. Protocol Wrappers for Layered Network Packet Processing in Reconfigurable Hardware. *IEEE Micro*, volume 22, 2002: pp. 66–74.

[14] Dedek, T.; Martínek, T.; Marek, T. High Level Abstraction Language as an Alternative to Embedded Processors for Internet Packet Processing in FPGA. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, Aug 2007, pp. 648–651.

[15] Kobiersky, P.; Kořenek, J.; Polčák, L. Packet header analysis and field extraction for multigigabit networks. In *Design and Diagnostics of Electronic Circuits Systems, 2009. DDECS '09. 12th International Symposium on*, April 2009, pp. 96–101.

[16] Attig, M.; Brebner, G. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, Oct 2011, pp. 12–23.

[17] Puš, V.; Kekely, L.; Kořenek, J. Design methodology of configurable high performance packet parser for FPGA. April 2014.

[18] Puš, V. *Packet Classification Algorithms*. Dissertation thesis, FIT BUT, 2012.

[19] Dharmapurikar, S.; Song, H.; Turner, J.; et al. Fast Packet Classification Using Bloom Filters. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-580-0, pp. 61–70, doi:10.1145/1185347.1185356. Available from: http://doi.acm.org/10.1145/1185347.1185356

[20] Gupta, P.; Mckeown, N. Packet Classification using Hierarchical Intelligent Cuttings. In *Hot Interconnects VII*, 1999, pp. 34–41.

[21] Singh, S.; Baboescu, F.; Varghese, G.; et al. Packet Classification Using Multidimensional Cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, New York, NY, USA: ACM, 2003, ISBN 1-58113-735-4, pp. 213–224, doi:10.1145/863955.863980. Available from: http://doi.acm.org/10.1145/863955.863980

[22] Qi, Y.; Fong, J.; Jiang, W.; et al. Multi-dimensional packet classification on FPGA: 100 Gbps and beyond. In *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 241–248, doi:10.1109/FPT.2010.5681492.

[23] Qi, Y.; Xu, L.; Yang, B.; et al. Packet Classification Algorithms: From Theory to Practice. In *INFOCOM 2009, IEEE*, April 2009, ISSN 0743-166X, pp. 648–656.

[24] Fong, J.; Wang, X.; Qi, Y.; et al. ParaSplit: A Scalable Architecture on FPGA for Terabit Packet Classification. In *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*, Aug 2012, ISSN 1550-4794, pp. 1–8, doi:10.1109/HOTI.2012.17.

[25] Srinivasan, V.; Varghese, G.; Suri, S.; et al. Fast and Scalable Layer Four Switching. *SIGCOMM Comput. Commun. Rev.*, volume 28, no. 4, Oct. 1998: pp. 191–202, ISSN 0146-4833, doi:10.1145/285243.285282. Available from: `http://doi.acm.org/10.1145/285243.285282`

[26] Eatherton, W.; Varghese, G.; Dittia, Z. Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *SIGCOMM Comput. Commun. Rev.*, volume 34, no. 2, Apr. 2004: pp. 97–122, ISSN 0146-4833, doi:10.1145/997150.997160. Available from: `http://doi.acm.org/10.1145/997150.997160`

[27] Michal Kekely. *Mapping of Match Tables from P4 Language to FPGA Technology.* Master's thesis, FIT BUT, 2016.

[28] Taylor, D. E.; Turner, J. S. Scalable packet classification using distributed crossproducing of field labels. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 1, March 2005, ISSN 0743-166X, pp. 269–280 vol. 1.

[29] Kekely, L.; Puš, V.; Kořenek, J. Software Defined Monitoring of application protocols. In *INFOCOM, 2014 Proceedings IEEE*, April 2014, pp. 1725–1733.

[30] Puš, V.; Benáček, P. Application specific processor with high level synthesized instructions. In *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*, 2014, p. 246.

[31] Open Networking Foundation. Software-Defined Networking (SDN) Definition. Available from: `https://www.opennetworking.org/sdn-resources/sdn-definition`

[32] Ryu SDN Framework Community. Component-Based Software Defined Networking Framework. Available from: `https://osrg.github.io/ryu/`

[33] Open Networking Foundation. OpenFlow Switch Specification version 1.5.1 (Protocol version 0x06). Available from: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf`

[34] Lavasani, M.; Dennison, L.; Chiou, D. Compiling high throughput network processors. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1155-7, pp. 87–96.

[35] Xilinx. Software Defined Specification Environment for Networking (SDNet). May 2014. Available from: `http://www.xilinx.com/support/documentation/backgrounders/sdnet-backgrounder.pdf`

[36] Brebner, G. P4 for an FPGA. June 2015. Available from: `http://schd.ws/hosted_files/p4workshop2015/33/GordonB-P4-Workshop-June-04-2015.pdf`

[37] Bianchi, G.; Bonola, M.; Capone, A.; et al. OpenState: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review*, volume 44, no. 2, 2014: pp. 44–51.

[38] OpenState SDN Project. OpenState SDN. Available from: `http://openstate-sdn.org/`

[39] Tinnirello, I.; Bianchi, G.; Gallo, P.; et al. Wireless MAC processors: Programming MAC protocols on commodity Hardware. In *INFOCOM, 2012 Proceedings IEEE*, March 2012, ISSN 0743-166X, pp. 1269–1277, doi:10.1109/INFCOM.2012.6195488.

[40] Bianchi, G.; Gallo, P.; Garlisi, D.; et al. MAClets: Active MAC Protocols over Hard-coded Devices. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1775-7, pp. 229–240, doi:10.1145/2413176.2413203. Available from: `http://doi.acm.org/10.1145/2413176.2413203`

[41] Bosshart, P.; Daly, D.; Gibb, G.; et al. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, volume 44, no. 3, July 2014: pp. 87–95, ISSN 0146-4833, doi:10.1145/2656877.2656890. Available from: `http://doi.acm.org/10.1145/2656877.2656890`

[42] P4 Language Consortium. P4. Available from: `http://p4.org/`

[43] Santiago del Rio, P. M.; Rossi, D.; Gringoli, F.; et al. Wire-speed Statistical Classification of Network Traffic on Commodity Hardware. In *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, IMC '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1705-4, pp. 65–72, doi:10.1145/2398776.2398784. Available from: `http://doi.acm.org/10.1145/2398776.2398784`

[44] Marian, T.; Lee, K. S.; Weatherspoon, H. NetSlices: Scalable Multi-core Packet Processing in User-space. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1685-9, pp. 27–38, doi:10.1145/2396556.2396563. Available from: `http://doi.acm.org/10.1145/2396556.2396563`

[45] Kohler, E.; Morris, R.; Chen, B.; et al. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, volume 18, no. 3, 2000: pp. 263–297.

[46] P4 Language Consortium. P4-HLIR. Available from: `https://github.com/p4lang/p4-hlir`

[47] P4 Language Consortium. P4C-BEHAVIORAL. Available from: `https://github.com/p4lang/p4c-behavioral`

[48] Giladi, R. *Network Processors: Architecture, Programming, and Implementation*. Systems on Silicon, Elsevier Science, 2008, ISBN 9780080919591.

[49] Netronome. Agilio CX Inteligent Sever Adapters. June 2016. Available from: `https://www.netronome.com/products/intelligent-server-adapters/agilio-cx/`

[50] NetFPGA. NetFPGA-10G Information. 2014. Available from: `http://netfpga.org/10G_specs.html`

[51] Xilinx. 7 Series FPGAs Memory Resources. November 2014. Available from: `http://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf`

[52] Xilinx. 7 Series DSP48E1 Slice. November 2014. Available from: `http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf`

[53] Xilinx. Zynq-7000 All Programmable SoC Overview. September 2016. Available from: `http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`

[54] Xilinx. RocketIO X Transceiver User Guide. February 2007. Available from: `http://www.xilinx.com/support/documentation/user_guides/ug035.pdf`

[55] Xilinx. 7 Series FPGAs GTX/GTH Transceivers. August 2015. Available from: `http://www.xilinx.com/support/documentation/user_guides/ug476_7Series_Transceivers.pdf`

[56] Gibb, G. *Reconfigurable Hardware for Software-Defined Networks*. Dissertation thesis, Stanford University, 2013.

[57] Postel, J. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), Sept. 1981, updated by RFCs 1122, 3168, 6093, 6528. Available from: `http://www.ietf.org/rfc/rfc793.txt`

[58] Puš, V.; Kekely, L.; Kořenek, J. Low-latency Modular Packet Header Parser for FPGA. 2012, doi:10.1145/2396556.2396571. Available from: `http://doi.acm.org/10.1145/2396556.2396571`

[59] Meiners, C. R.; Liu, A. X.; Torng, E. *Hardware Based Packet Classification for High Speed Internet Routers*. Springer Publishing Company, Incorporated, first edition, 2010, ISBN 1441966994, 9781441966995.

[60] Kekely, L.; Kučera, J.; Puš, V.; et al. Software Defined Monitoring of Application Protocols. *IEEE Transactions on Computers*, volume 65, no. 2, Feb 2016: pp. 615–626, ISSN 0018-9340, doi:10.1109/TC.2015.2423668.

[61] Moore, A. W.; Zuev, D.; Crogan, M. L. Discriminators for use in flow-based classification. Technical report, 2005.

[62] Piskač, P.; Novotný, J. Using of Time Characteristics in Data Flow for Traffic Classification. In *Managing the Dynamics of Networks and Services*, *Lecture Notes in Computer Science*, volume 6734, edited by I. Chrisment; A. Couch; R. Badonnel; M. Waldburger, Springer Berlin Heidelberg, 2011, ISBN 978-3-642-21483-7, pp. 173–176, doi: 10.1007/978-3-642-21484-4_21. Available from: `http://dx.doi.org/10.1007/978-3-642-21484-4_21`

Reviewed Publications of the Author Relevant to the Thesis

Reviewed publications

[A.1] P. Benáček, V. Puš and H. Kubátová. *P4-to-VHDL: Automatic Generation of 100Gbps Packet Parsers.* IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM2016), Washington D.C., USA, 2016.

[A.2] P. Benáček, V. Puš and H. Kubátová. *Automatic Generation of 100 Gbps Packet Parsers from P4 Description.* First International Workshop on Heterogeneous High-performance Reconfigurable Computing, Austin, TX, USA, 2015.

[A.3] P. Benáček, H. Kubátová and V. Puš. *Architecture of Effective High-Speed Network Stream Merger.* Digital System Design (DSD), 17th Euromicro Conference on Digital System Design, pp. 459–464, Verona, Italy, 2014.

[A.4] P. Benáček and V. Puš. *Application specific processor with high-level synthesized instructions.* ACM/SIGDA international symposium on Field-Programmable Gate Arrays, pp. 246–246, Monterey, CA, USA, 2014.

[A.5] P. Benáček, T. Čejka, H. Kubátová and R. Blažek. *Change-Point Detection Method on 100 Gb/s Ethernet Interface.* ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Marina del Rey, CA, USA, 2014.

[A.6] P. Benáček, T. Čejka, H. Kubátová, L. Kekely and R. Blažek. *FPGA Accelerated Change-Point Detection Method for 100 Gb/s Networks.* Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, Telč, Czech Republic, 2014.

[A.7] P. Benáček, V. Puš and P. Kaštovský. *P4-to-FPGA: High Performance Reconfigurable Networking (poster).* Second International Workshop on Heterogeneous High-performance Reconfigurable Computing, Salt Lake City, UT, USA, 2016.

[A.8] P. Benáček and V. Puš *P4-to-VHDL: Generating High Speed Network Devices (poster).* P4 Workshop, Stanford, California, USA, 2016. Available from: `https://www.liberouter.org/wp-content/uploads/2016/06/poster_p4_06_2016.pdf`

[A.9] L. Kekely, V. Puš, P. Benáček and J. Kořenek. *Trade-offs and progressive adoption of FPGA acceleration in network traffic monitoring.* Field Programmable Logic and Applications (FPL), 24th International Conference, pp. 1–4, Munich, Germany, 2014.

The paper has been cited in:

- D. Grochol, L. Sekanina, M. Žádník, J. Kořenek. *Evolutionary circuit design for fast FPGA-based classification of network application protocols.* Applied Soft Computing, Volume 38, January 2016, pp. 933-941.
- L. Tang, J. Yan, Z. Sun, T. Li, M. Zhang. *Towards high-performance packet processing on commodity multi-cores: current issues and future directions.* Research Paper Special Focus On Future Internet Architecture And Protocol Science China Information Sciences, December 2015, Volume 58, Issue 12, pp. 1-16.

Submitted publications

[A.10] P. Benáček, H. Kubátová and V. Puš. *P4-to-VHDL: Automatic Generation of High-Speed Input and Output Network Blocks.* Microprocessors and Microsystems, Elsevier Journal, 2016.

Remaining Publications of the Author Relevant to the Thesis

[A.11] P. Benáček. *Methodology for Analysis of High-Speed Networks.* Ph.D. Minimum Thesis, Faculty of Information Technology, Czech Technical University in Prague, Czech Republic, 2014.

[A.12] P. Benáček, H. Kubátová and V. Puš. *P4-to-FPGA: Translating P4 to VHDL.* The 4th Prague Embedded Systems Workshop (PESW 2016), ISBN 978-80-01-05984-5, Roztoky, Czech Republic, 2016.

[A.13] P. Benáček and V. Puš *The P4 Language as the Future of SDN (CZ).* CESNET blog, 2016. Available from: `https://www.cesnet.cz/2016/01/jazyk-p4/`

[A.14] P. Benáček and V. Puš *The P4 Language as the Future of SDN (CZ).* root.cz (part 1 and part 2), 2016. Available from: Part 1 - `http://www.root.cz/clanky/jazyk-p4-jako-budoucnost-sdn/`, Part 2 - `http://www.root.cz/clanky/jazyk-p4-jako-budoucnost-sdn-dokonceni/`

[A.15] P. Benáček. *Analýza rychlých síťových přenosů.* Česko-slovenský seminář PAD, pp. 9–12, ISBN 978-80-01-05106-1, Milovy ve Žďárských vrších, Česká republika, 2012. *The paper was awarded as the best publication of first-year students.*

[A.16] P. Benáček. *Architektura pro měření v reálném čase na vysokorychlostních sítích.* Česko-slovenský seminář PAD, pp. 45–50, ISBN 978-80-261-0270-0, Teplá, Česká republika, 2013.

[A.17] P. Benáček. *Real-time Network Measurement for High-Speed Networks.* The 1st Embedded Systems Workshop (ESW), Temešvár, Czech Republic, 2013.

Remaining Publications of the Author

[A.18] P. Benáček and M. Novotný. *Implementing Brute-Force Attack on PRESENT Cipher*. Digital System Design (DSD), 13th Euromicro Conference on Digital System Design, pp. 51–52, Linz, Austria, 2010.

[A.19] P. Benáček, T. Čejka, Š. Friedl and R. Krejčí. *COMET - COMBO Ethernet Tester*. Annual report, CESNET z.s.p.o., Prague, Czech Republic, 2013. Available from: `http://www.cesnet.cz/wp-content/uploads/2013/03/comet.pdf`

Evaluation Activities

[A.20] Conference Programme Committee member in the *Track on Reconfigurable Computing for Networks and Communications.* International Conference on Reconfigurable Computing and FPGAs (ReConFig'2016), Cancum, Mexico, 2016. Available from: `http://www.reconfig.org/index.php?option=com_content&view=article&id=6&Itemid=134`

[A.21] Review of article for the *Journal of Parallel and Distributed Computing.* Article in journal, ISSN 0743-7315, 2016. Available from: `http://www.journals.elsevier.com/journal-of-parallel-and-distributed-computing`