HP4 High-Performance Programmable Packet Parser

Amr Ibrahim[1, 2]

[1]Systems Engineering and Computer Department, Faculty of Engineering, Alazhar University, Cairo, Egypt
[2]Computer Science Department, College of Computing and Information Technology, University of Bisha, Bisha, KSA

***Abstract*:** Now, header parsing is the main topic in the modern network systems to support many operations such as packet processing and security functions. The header parser design significantly affects the network devices' performances (latency, throughput, and resource utilization). However, the header parser design suffers from many difficulties, such as the incrementing in network throughput and various protocols. Therefore, the programmable hardware packet parsing is the best solution to meet the dynamic reconfiguration and speed needs. Field Programmable Gate Array (FPGA) is an appropriate device for programmable high-speed packet implementation. Most high-speed programmable packet systems use the P4 language (Programming protocol-independent Packet Processors) because it is a high-level abstract description language. This paper introduces a novel FPGA High-Performance Programmable Packet Parser architecture (HP4). HP4 is automatically generated by the generating unit (convert P4 programs to VHDL code) to optimize the speed, dynamic reconfiguration, and resource consumption. The HP4 shows a pipelined packet parser dynamic reconfiguration and low latency. In addition to high throughput (over 600 Gb/s), HP4 resource utilization is less than 7.5 percent of Virtex-7 870HT, and latency is about 88 ns. High-speed dynamic packet switch and network security can use HP4.

***Keywords*:** Programmable Packet Parsers, Pipeline, Latency, Throughput, Resource Utilization, FPGA, P4.

1. Introduction

Recently, computer networks have evolved in both speed and variety of protocols (number and types). Therefore, there is a need for a prodigious packet parser at all modern network infrastructure [1]. However, many problems are facing the design and the implementation of the parser such as (1) processing at a line rate in the high-speed network (parsing millions of packets per second), (2) adaptation to new protocols; the number and types of protocol types are varied (adding a new protocol needs an experienced designer acclimated to the HDL language or parser architecture), (3) incomplete information (some protocols have more one format: standard and customized), (4) the header fields attributes (number, size, and location) varied with the protocol type, (5) the parser must have a small size because of the restriction of the programmable device's size, and (6) the enormous hole between the product description and the hardware implementation in the device of new types protocol. These problems demand a programmable hardware packet parsing [2], [3].

Programmable packet parser relies on three steps: (1) high-level protocol description, (2) automatic code generation, (3) dynamic reconfigurations. Therefore, the proposed system is a High-Performance Programmable Packet Parser (HP4). HP4 used the P4 language to describe protocols and include it in FPGA as a target platform. P4 (Programming Protocol-independent Packet Processors) is the de facto standard high-level language for describing packet protocols and rules for headers parsing at runtime. Recently, the P4 has gained adoption in academia and manufacturing [4], [5]. It has two versions. P416 released in 2017 with a new feature to overcome the limitation of P414 [1]. HP4 used P416. Generally, P4 has many advantages, such as protocol independence, fields' reconfiguration, and portability, and free and open-source tools [6], [5]. The programmer decides how the forwarding plane processes packets without stressing over the implementation details. After that, he can do a converting the P4 description program to a suitable synthesizable VHDL code for FPGA and ASCII platforms. So P4 enables a new generation of networking hardware programming that can be dynamic reconfiguration and independent target. FPGAs are the best target platform for P4 programs at hardware line rates [1], [5], [6]. FPGA is a complete framework on a chip, including memory blocks, multiplier, accumulator units, and embedded processors. It is the most elegant solution for implementing a reprogrammable network system due to its performance: simplicity, speed of reconfiguration, low power utilization, and high performance. FPGA includes utilizing low-level hardware description languages (i.e., VHDL, Verilog) [7], [4]. Xilinx ISE presents a number of the synthesis tools, libraries, and simulation that help in architecture synthesis in Xilinx devices [15]. It recommends the Vivado as a High-Level Synthesis for the new versions of Xilinx devices (e.g. Virtex-7) [16]. Virtex-7 870HT FPGAs can accomplish higher throughput parser (400 to 800b/s), bring down latency, and reduce power consumption [7]. Now, major cloud providers, such as Microsoft, Amazon, and Baidu, convey FPGAs in their data centers to help execution, e.g., accelerate network encryption and decryption or implement custom transport layers [8]. Because of the P4 and FPGA advantages, HP4 used the P4 and FPGA in the design and implementation phases.

This work aims to design and implement HP4 based on FPGA to solve some of the drawbacks of prior works, including trading-off architecture, high-speed wire, latency, and resource usage. HP4 uses the issues that are marked with the red and italic font in Figure 1 with hypotheses of the maximum of (1) frequency, (2) word width (3) protocols stack size.

HP4 architecture is a streaming packet parser; thus, its operations start once to receive the data from the data bus. It uses pipelining to achieve a high throughput and processing chain to represent the incoming network packet's protocol stack. It also uses multiple parsers in parallel to increase the

speed by pipelines. The increase in enabled pipelines led to increasing the frequency, and the throughput will rise. However, latency and consumed resources will increase. The number of parser stages is optional. HP4 adjusts between it and both latency and resource utilization to find the optimal parameters. The ingress and egress pipelines are shared across the input-output interface to reduce the chip area. HP4 is generated and optimized automatically for resource utilization, latency, and throughput.

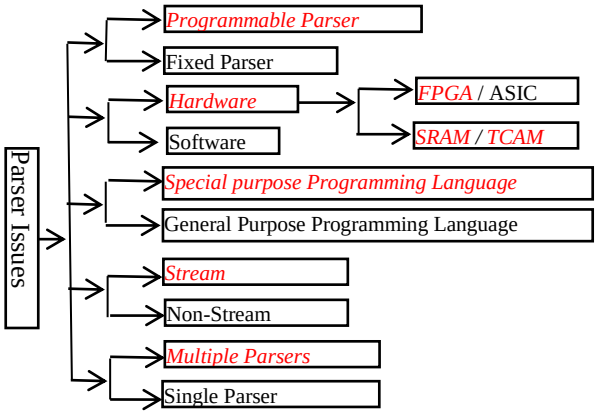

Figure 1. The Parser Design Issues

The generator unit converts the parse graph description of P4 code to a VHDL code adequate for implementing in Xilinx Virtex-7 870HT FPGA. This paper presents a novel design, parsing a raw throughput in FPGA with more than 600 Gb/s at the line rate with less than 7.5 percent of available resources. The implementation included too word width (up to 4096-bit) parallel data buses for streaming packet data through pipeline stages. The results show the scalability, resource utilization, throughput, and latency of HP4, for different widths of the data bus and the number of pipeline stages.

The rest sections of this paper are: Section II provides other related works. Section III presents the data bus structure and HP4 abstract architecture. Parse Graph Representation (PGR) description and optimization, P4 language, and how to generate the parser components and convert them to VHDL (synthesis) are in section IV. Section V presents the validation and evaluation of HP4. Finally, section VI debates the conclusions.

2. Related Work

Kozanitis et al. [10] presented the Kangaroo system to parse several protocols header in one-step used CAM to extract the next bytes. Its throughput was about 40 Gb/s line rate and used less than 1% of the chip area with 400 MHz ASIC. It buffered all header fields before the parsing (non-streaming), so the latency was too high. Yang and Prasanna showed in [11] IP address lookup could up to 100 Gb/s rates using FPGA. However, the two systems had problems, such as storing and accessing the long packets in the memory. Hence, the CAMs becomes a bottleneck because it has some limitations compared to SRAMs such as low storage capacity, slow access time relatively, low scalability, and highly expensive.

Gibb et al. [13] provided the design details of a fixed and configurable packet parser. This work did not show FPGA implementation results. They assumed ASICs as the target implementation platform. There are many approaches to FPGA packet parser published with many advantages and disadvantages [9]. Puš et al. [2] proposed a hand-optimized pipelined packet parser. It used only 1.19% of the Virtex-7 870HT FPGA to achieve throughput over 100 Gb/s and 4.88% for throughput over 400 Gb/s with reasonable latency. However, this parser is only enough for wire-speed up to 100 Gb/s with 512-b data bus width [9].

Attig and Brebner in [12] presented their Packet Parsing language (PP) to describe the packet headers and the parse graphs. They used several heavily pipelined templates based on the Yang and Prasanna works [11] to parse 400 Gb/s on a single Xilinx Virtex-7 870HT FPGA. The throughput is affected by the shortest frames (so only up to 100 Gb/s). However, the PP language ignored the identification of the packet flow control.

Benácek et al. [6] offered an automatic P4-to-VHDL packet parser generator based on Xilinx Virtex-7 XCVH580T FPGA. The generated parser worked with 100 Gb/s with roughly 100% overhead in terms of latency and resource consumption compared to a hand-written VHDL implementation [4] [7]. Wang et al. [8] introduced a quick framework of 10 Gb/s parser without architecture details for generating VHDL code from the P4 programs [9]. Jakub et al. in [9] produced auto-generated parsers with throughput over 1 Tb/s on the Xilinx UltraScale FPGAs and about 800 Gb/s on Virtex-7 FPGAs. They used P4 language, multiple pipelines, and parallel packet parsing combining by multiple packets per one data frame [1]. Silva et al. [4] presented 100 Gb/s open source pipelined streaming packet parser based on FPGA Virtex-7. They improved the pipeline structure and used two languages: C++ in the parser's specification and production of RTL (Register Transfer Language) code, and P4 in the description and optimization of PGR. This parser achieved low-latency and high-speed, but its logical resource utilization is high [7]. Cao et al. [1] validated a pipeline-based parser of both the full and simple types with throughputs of 358 Gb/s and 317 Gb/s. They presented an approach to convert P416 programs into VHDL and implemented it in FPGAs automatically. Lixin et al. [7] presented SDPIP parser (software-defined protocol independent) based on Virtex-7 FPGA. It had a 256 b data bus and throughput about 80Gb/s.

3. Architecture

3.1 Data Bus Structure

Data bus width is a necessary factor in implementing a parser, especially in high bandwidth systems and the low FPGA frequency.

HP4 modified the data bus structure mentioned in [9] to minimize the overload size. HP4 divided the data bus structure (Word) into many sections with a fixed size equal to the minimum Ethernet packet size (512 bit). The maximum number of packets per one-word with 4096 bits is eight sections, and eight parallel copies of the pipeline. The number of sections is a variable to manage in the number of transmitting packets per clock. The total number of sections is N= 2n, where n= 0,1,2,3, and N's default value is eight. The data bus width (w) equals N times 512b. Each section has sixteen logical partitions (N=16*P). Each partition (P)

composes of two items. Item size (I) is the smallest distinct piece of bits (16-bit). The new frame must start with a part and finish at any position. This structure of the data bus controls the alignment overhead between every two frames. The alignment overhead in [9] per packet was not over 7 bytes, but in HP4 is less than two bytes.

HP4 allows with three types of alignment to develop the bandwidth raw, as shown in Figure 2.

1) Full word: One packet per word as frame 1.
2) Share words: A word may contain many packets (e.g., frames 2 and 3 ends in the middle of a part).
3) Partial word: The packet may not overlap within the word, and the partially aligned start condition not violates as frame 4.

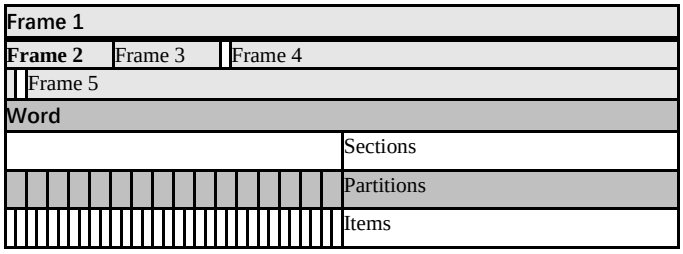

Figure 2. HP4 Data Bus Structure (modified structure [9]).

3.2 HP4 Parser Architecture

HP4 has general architecture enough for parsing of most protocols. The functions of HP4 are (1) receive the header data in a stream, (2) check and identify the protocols types, and (3) extract the header fields to generate the packet header vector (PHV) contains header fields classified by the protocol. Figure 3 shows an overview of HP4.

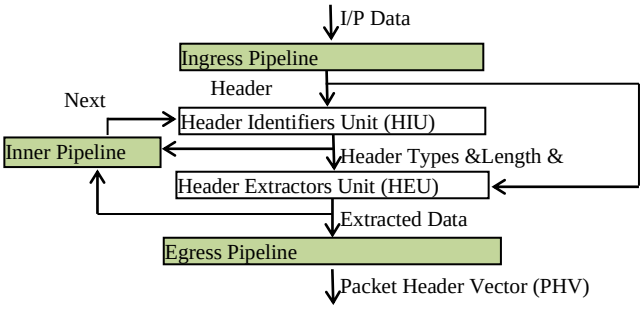

Figure 3. The Overview of HP4 Parser Model

3.2.1 Header Identifiers Unit (HIU)

Header Identifiers Unit (HIU) is the heart of HP4. Its functions are: (1) Receive the input data stream from the ingress pipeline, (2) Identify the protocols types, the length of the header, and the fields' location in the packet with the help of the parse graph (HIU depends on PGR generated by P4), and (3) Send these identifications to the Header Extractors

HIU consists of three principal identifiers modules. (1) Next Protocol Identifier determines the expected next protocol type. 1) Next Protocol Identifier, determines the expected next protocol type. It converts some of the extracted header bytes into an internal code representing the next protocol type.

Internal code is a unique identifier code for each header type. (2) Header Length Identifier determines the length of the current protocol header by computing the number of extracted bytes. (3) Field Location Identifier defines where the fields in the packet. It determines the sum of the current header offset (a value from ingress pipeline; Current Offset) and the current header length (the output of the Length Identifier).

Figure 4 shows the abstract architecture of the HIU. It contains a set of state machines (searching engine), buffers, and matched tables stored in TCAM and SRAM. TCAM stores the input data stream. The search engine searches in the match tables and returns the first matching entry (Protocol-Code). Then Protocol-Code is sent to the SRAMs to know the type and the length and generates Next-Header-Valid to propagate among the blocks within the same clock cycle. Algorithm1 shows the generation of Header Identifications.

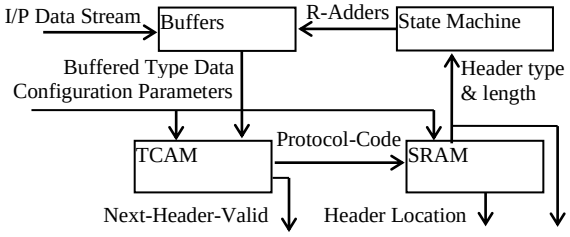

Figure 4. The Abstract Architecture of the Header Identifier Unit (HIU)

Algorithm 1: Header Identifiers

```
Function HeaderIdentifier (InputDataPacket)
Input: InputDataPacket
Output: HeaderIdentifications (HeaderType, HeaderLen,
        HeaderLoc)
begin
   HeaderType = InitialType;
   Location = 0;
   while (headerStack) do
       FieldExtraction (HeaderType; HeaderLoc)
       HeaderType = GetNextHeaderType (packet; HeaderType;
           Location)
       HeaderLen = GetHeaderLen (packet; HeaderType;
           HeaderLoc)
       HeaderLoc = Location + Length;

   return (HeaderLen, HeaderType; HeaderLoc)
end
```

3.2.2 Header Extractors Unit (HEU)

The functions of HEU are (1) read the header type, length and the location from HIU, (2) extract the header fields, (3) generate the PHV, and (4) send this vector to the egress pipeline.

Figure 5 shows the abstract architecture of the Header Extractors Unit. The buffers store the packet data while waiting for the header identifications (type, length, and location) from the HIU. Once it receives these identifications, it will extract the fields by multiple parallel headers extractors. The header extractors, header identifiers, and their internal elements run in parallel with minimum data dependency.

HEU unit determines for each byte in the input word if it will extract or discard. HEU depends on the state machine's output based on the configuration parameters and header identifications being input. Bytes, which were marked to extract, will be added to the memory blocks' position in the output stream. The memory blocks (a wide array of FIFO

registers) accumulate the extracted header fields into the PHV. FIFO is a 4Kb RAM with four dual ports. HEU also contains a crossbar switch that uses programmable and dynamic setting multiplexers (one per register). The crossbar switch helps extract data from any byte position and select between each output field to optimize resource utilization.

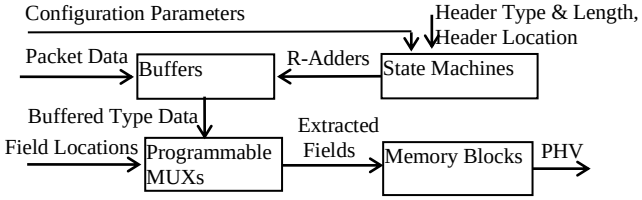

Figure 5. Abstract Architecture of the Header Extractors Unit (HEU)

It uses the parameter of the location as an address. Algorithm 2 illustrates the generation of the extracted fields.

Algorithm 2: Header Extractor

```
Function ExtractorFields (InputDataPacket; HeaderIdentifications)
Input: InputDataPacket, HeaderIdentifications
Output: ExtractedFields
begin
    Fields = GetFieldList(header)
    for (FieldPos; FieldLength)
      Fields = Extract (packet; HeadedrLocation + FieldLocation;
          FieldLength)
  return (ExtractedFields)
end
```

3.2.3 Pipeline Units

HP4 uses the pipeline to do a chain of parsing stages to parse the protocol stack and arrange the parser units. The pipeline functions are: (1) enter the essential data into and through HP4 to parse the current header protocol, (2) output the required information for the next header parsing stage or to the output, and (3) interface between the parser and the other elements. The pipeline helps develop by adding additional modules to parse new protocol type or reusing modules without any modification in the parser's internal structure. Several modules are already usable, and most of the parser modules for more protocols are almost identical.

There are three types of the pipeline, as shown in Figure 6: (1) Ingress pipeline: it generates the initial required data for the first protocol type stage. (2) Inner pipelines: it arranges the parser units and connects between each two pipeline stages. It is optional (enabled/ disabled) at the run time for each protocol type (stage) individually. (3) The egress pipeline passes the results to the output. The output pipeline plays the role of a bypass unit. The pipeline consists of many stages and Stage-Selector. The Stage-Selector receives the next protocol type, next header valid, and the configuration parameters to generate the Selector signal. The Selector is used to enable the suitable stage from the inter pipelines; otherwise, select the egress pipeline. The order of protocol parsers stages in the pipeline depends on the protocol stack. Each stage contains at least one header identifier unit and header extractor unit to represent one protocol type of the header stack. Multiple copies perform multiple parsers per stage. The number of parsers per stage is equal to the number of sections (N) per word. P4 generates the pipeline unit automatically. Algorithm 3 describes the pipeline process.

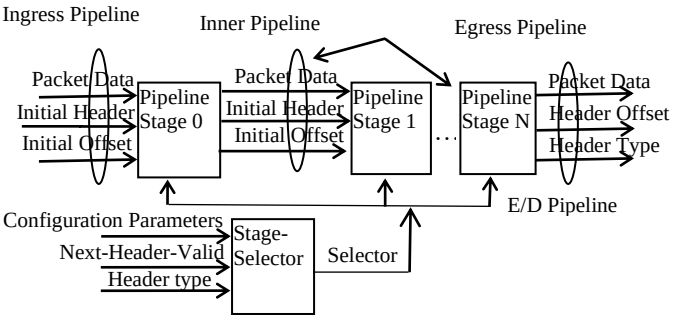

Figure 6. The abstract structure of the Packet parsing pipeline

Algorithm 3: Pipeline

```
Procedure Pipeline (packet; ProcessingChain)
  Input: packet, ProcessingChain
  Output: PacketHeaderVector
  begin:
    do
      /* Generate the Header Identifier Unit*/
      HeaderIdentifications = HeaderIdentifier (InputDataPacket);
      /* Generate the Header Identifier Unit*/
      ExtractedFields =HeaderExtractor (InputDataPacket,
      HeaderIdentifications);
      Selector = StageSelector (NextProtocolType;
          NextHeaderValid)
    while (Selector; PipelineStages; HeaderStack)
    /* Output the packet header vector PHV */
    PHV = ExtractedFields;
  end
```

4. HP4 Design and Implementation

4.1 Parse Graph Representation (PGR).

The Parse Graph Representation (PGR) is an acyclic oriented graph generated by the P4 from header description. There are two types of parse graph (1) fixed parse graph does not change after the implementation. It used in a fixed parser (non- programmable), and (2) programmable runtime parse graph (our work).

Figures 7 shows an example of a parse graph for 2xVLAN, IPv4, IPv6, TCP, UDP, ICMP, ICMPv6. Each graph node (state) represents one header type. Starting with the root node, and each edge (leaf) represents the next protocol type. Each path is a header sequence. The topological ordering of PGR nodes depends on the Depth-First Search (DFS) algorithm. The P4 packet parser description program also defines the condition of a transition. If the state is not described in the P4 program and required by the parser, it will be the infinite state, and the transit state is the terminate (end) state. Loop edge can represent the situation to support more protocols of the same protocol type in the protocol stack (like the two VLAN). HP4 will translate each node into VHDL code automatically, and an optional pipeline will separate between every two nodes.

The represented parse graph in Figure 7 is unsuitable for a fast parser because it contains many paths. Then it will require many bypass pipelines. The increase in the pipeline number tends to a series of delays and increases in resource usage. HP4 optimizes this graph to present a suitable PGR for the high-speed network. The steps of the parse graph optimization are:

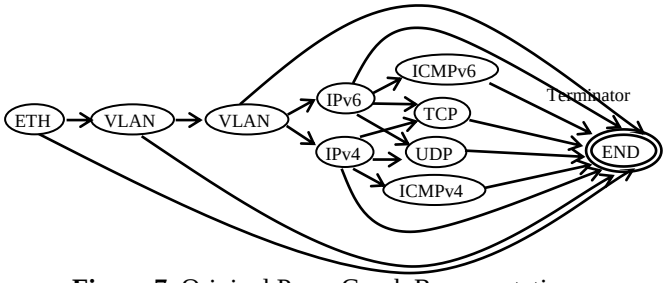

Figure 7. Original Parse Graph Representation

(1) get the original parse graph representation from the compilation of the P4 program description as shown in Figure 7, (2) determine the level for each node by computing the path length (distance from the root node to each node), (3) find out the longest paths of the parser graph, (4) eliminate and delete the redundant paths (the children's paths) for each longest path, and then (5) generate the optimized parse graph as shown in Figure 8.

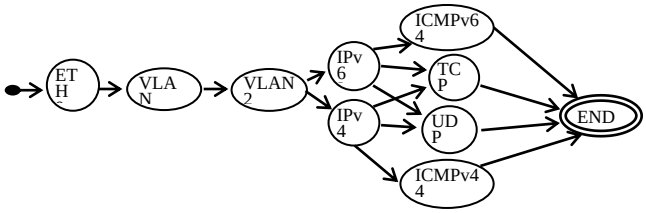

Figure 8. Final optimized Parse graph

4.2 P4

P4 utilized to describe each header field (name, size in bits, and the sequence), and the extract fields (type, length, and location), and define the constraints of the field size. Because the field size can be constant or variable, the header length may be fixed (the length is the sum of all field sizes) or variable length. The problem in the variable header field length where its value is nondeterministic. P4 solves this situation by defining the header length as a variable. Then the variable header length is a formula calculates the derived fields' sizes from the input data raw at the runtime like the following [14]:

```
header_type ethernet {
                fields {
                        dst_mac : 48;
                        src_mac : 48;
                        ethertype : 16;
                        }
                }
header_type ipv6_ext_t {
    fields {
            nextHdr : 8;
            totalLen : 8;
            frag : 12;
            padding : 3;
            fragLast : 1;
            }
    length : (totalLen + 1) * 8;
    max_length : 1024; // Bytes
        }
header_type vlan_t {
                fields {
                pcp : 3;
                cfi : 1;
                vid : 12;
                ethertype : 16;
                }
                    }
header_type udp_t {
        fields {
                srcPort : 16;
                dstPort : 16;
                length : 16;
                checksum : 16;
                }}
```

P4 presents the transition in the pipeline states by a switch-case command in a tuple form (value, next state) and extract command to parse the fields and constructs the parse graph. The switch case command uses the extracted data to present the next state (next protocol type). It determines which subset of the defined header types could occur at each pipeline stage. If the header is unknown, the P4 uses the default ingress to drop the packet, go to the end state, and start a new ingress. The extract instruction determines fields will extract. The return ingress command indicates that the current parsing stage is complete. It returns the next parsing stage to the start and demonstrates the beginning of the pipeline, for example [14]:

```
header ethernet eth;
        parser ethernet {
                extract(eth);
                switch(eth.ethertype) {
                        case 0x8100: vlan;
                        case 0x9100: vlan;
                        case 0x800: ipv4;
                        default: ingress;
                        }
        }
        extract(udp);
        return ingress;
```

The P4 program is also used to define the three HIU's identifiers as an object's method for running at runtime, the header fields, and the needed information for the identifiers. The StageSelector method checks the current parsing stage's output, decides to allow a new pipeline stage or considers it the last, and passes it to the output stage.

4.3 HP4 Implementation

The P4 compiler converts the HP4 abstract architecture from the P4 program to the VHDL code using the code generation unit. Xilinx Vivado 2018.2 design suite tool synthesized this VHDL code to Xilinx Virtex-7 870HT FPGA. The HP4 design is then synthesized with the various setting parameters (data bus width and number of pipeline stages), placed and routed, and generated a Bitstream (physical FPGA programming information). The automatic generation of code has many advantages, such as 1) the simplicity of generating VHDL code, and 2) the modularity to allow by the extension in two ways: (i) adding externally defined features and (ii) changing or removing the basic block.

HP4 uses two types of memories (TCAMs and SRAM) in the implementation. TCAM stores bit sequences to perform the longest prefix and parallel match to identify headers. SRAM stores the next state information, extracted fields, and any other data needed during the parsing to perform the hash-based binary match and the exact match rapidly. The field buffer design is flexible: it is a set of registers (array) to help the extracted fields transmit in a parallel to downstream components. HP4 minimizes the number of the extracted fields to reduce the field buffer size and resource utilization. HP4 uses a 4Kb packet header vector during parsing.

There many steps to implement HP4: (1) Writing the P4 program to describe the protocols. (2) Creating the PGR from the P4 program. (3) Optimizing the PGR by

eliminating the children's paths as in Figure 8. (4) Creating the processing chain from the PGR and pipeline stages. Each node contains a number represent the node level (stage number). Some levels have more than one node on the same level; then, several parsers will connect in serial with the ordinary ordering. The analyzers of the same level connect in parallel. The serial connection allows HP4 to keep the processing chain's homogeneous structure, as shown in Figure 9. (5) Generating the VHDL code of the Header Identifiers Unit and Header Extractor Unit. (6) Creating and identifying the configuration parameter and the essential control elements in generating the parser units (such as the window size, the number and size of the parsed lookup table, and the programmable memory TCAM/RAM) size. Algorithm 4 demonstrates the generation of HP4 from the P4 program.

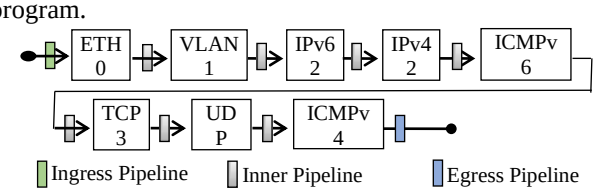

Figure 9. Generated processing chain

Algorithm 4: Generating the HP4 (Transformation from P4 to HP4

```
Procedure HP4Transformation(P4Program)
Input : P4Program
Input : InputDataPacket
Input: configuration-parameters
Output: HP4 VHDL Code
begin:
  /* Get the Parase Grapgh Represntation*/
  ParseGraph= ReadParseGraphRepresentation(P4Program);
  /* Discover the node levels */
  NodeLevel= GetNodeLevel(ParseGraph);
  /*Find out the longest path*/
  LongestPath= FindLongestLevel (ParseGraph, NodeLevel);
  /* Optimize the parse grapgh*/
  OptimizedParseGrapgh= EliminateChilderenPaths (PraseGrapgh,
        NodeLevel, LongestPath);
   /* Generate the Pipeline Stages*/
  PipelineStages= GenerateProcessingChaine
        (OptimizedParseGrapgh, InputDataPacket);
  Pipeline (packet; ProcessingChain)
  end
```

4.4 HP4 Optimization

HP4 supported many optimizations, saving significant resource utilization, reduced latency, and increased throughput. HP4 achieves high throughput in high-speed network processing by working in two dimensions. (1) Increasing the input data bus width to process multiple packets or partial packets per one clock cycle. The throughput depends on the word width and the expected clock frequency. (2) Maximizing the header stack size to increase the number of pipeline stages. The user supplies these two parameters to the P4 compiler and setting at the runtime.

The ingress and egress pipelines are the same in a physical block and the field allocation function for the optimization. HP4 also aggregates multiple parallel small packet parsers in each pipeline stage to parse one packet per the stage for each type of protocol in the stack. A single parser instance's processing affects latency and resource utilization for two reasons: (1) the packet data bus increases in width and needs more resources, and (2) additional headers can occur within a single processing data bus's section, requiring more header-specific processor instances for parsing processing. The following section will demonstrate these reasons. HP4 achieved throughput more than 600 Gb/s by grouping 16 instance parser of 40Gb/s running at 1 GHz.

5. Test and Evaluation

HP4 parser tested the design properties with two separate protocol stacks: (1) Full parse: 4×VLAN, 4×MPLS, IPv4 or IPv6 (2×extension headers), TCP or UDP, and (2) Simple parse: IPv4 or IPv6, (2×extension headers), TCP or UDP. The two protocol stacks were described in a P416 language and synthesized with various data bus width settings (512, 1024, 2048, and 4096 bits) and pipeline stages. In these cases, the data bus parameter allows adequate and efficient wire-speed parsing even for the smallest packets (512 bits) and the large packets with a varying N as specified in section (III.A). The setting of data bus width (N sections) and the number of pipelines tend to a wide range of possible outputs. From the different data bus width settings and the number of pipeline stages with the two protocol stacks, the HP4 parser state-space is different in throughput latency and resource use. The FPGA resource utilization contains two parts: resources consumed by the shared fixed runtime functions of all P4 programs, and the unique P4 components resources, which varied and dependent on the parameters. FPGA resource is the sum of the used LUTs and registers.

The results obtained by Vivado after synthesis of the Xilinx Virtex-7 870HT FPGA are throughput, latency, and resource use. HP4 evaluation occurred in two stages. Firstly, HP4's ability to manage a varied range of P4 parsers. Secondly, the generated parser can work online, using a collection of testbench circuits, with different setting parameters and the two protocol stacks (Full and Simple parses). The generator produced testbench circuits to verify the parser. Finally, in summary, Pareto's principle checks the results with the various configuration parameters to find the optimal solution for the HP4.

The following sections illustrate the HP4 test cases (Simple and Full parse) by three graphs per case. These graphs show the relationship between (1) throughput and resource utilization, (2) throughput and latency, and (3) resource utilization and latency. In addition to two graphs with Pareto sets for the two parse types show the optimization for: (1) the throughput and resource utilization, and (2) the throughput and latency.

5.1 Test

5.1.1 Simple Parser Test

Figure 10 shows the resource utilization with the throughputs for the Simple parser. From the graph, FPGA resource utilization linearly increases with the achieved throughput. In addition to doubling the data bus width does not double the FPGA resource utilization because there is a consumed resource in the computing components and the fixed functions. Figure 11 shows the latency and throughput for different settings of the Simple parser. From the graphs, the latencies are increasing as the achieved throughput is rising, because the high throughput requires more extensive registering. Generally, the latency depends on the configured

number of registers in the enabled pipeline stages and the working frequency.

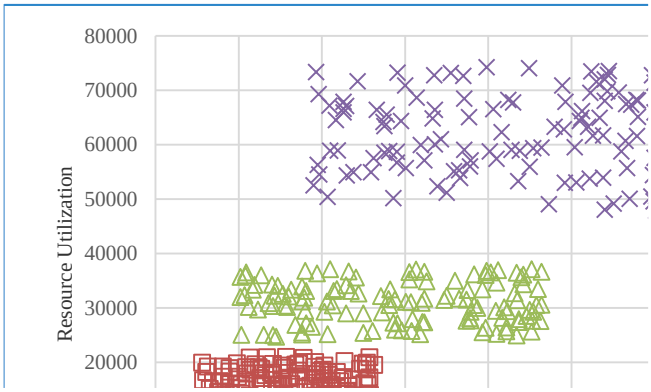

Figure 10. The relationship between Throughput and Resources of the Simple parser with a different data bus width

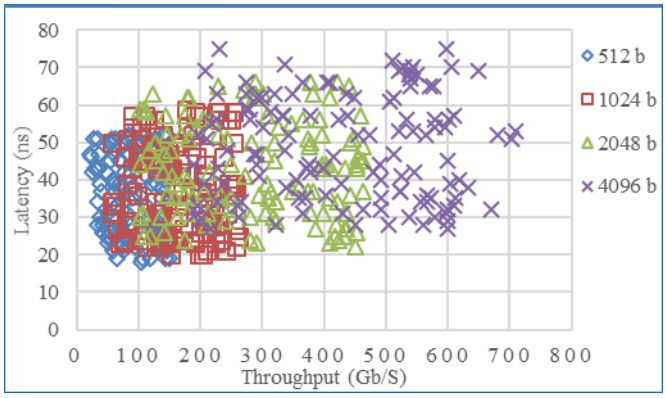

Figure 11. The relationship between Throughput and Latency of the Simple parser with a different data bus width

Figure 12 shows the relation between the latency and the resource utilization of the Simple parsers. The resource utilization rises considerably with the data bus width from the graph, and the latency pretty much stays in the same boundaries.

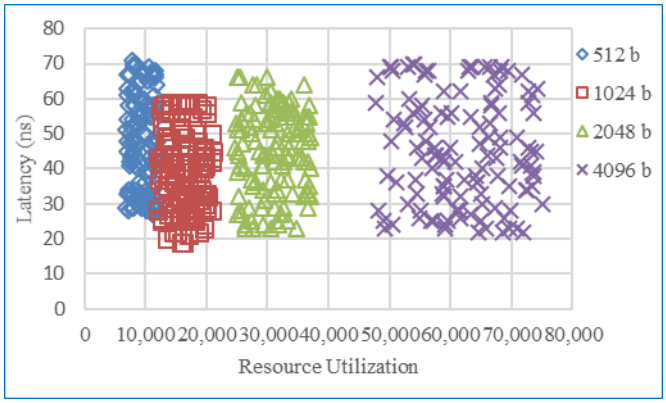

Figure 12. The relationship between Resources and Latency of the Simple parser with a different data bus width

5.1.2 Full Parser Test

The Full parser is much larger than the Simple parser in the size because it parses many protocols. The Full parser's state-space with the combinations of the set parameters is enormous for each word width. Therefore, HP4 synthesized only some hand-picked and randomly selected configurations of the possibilities. Figure 13 shows the resource utilization and effective throughput of the synthesized Full parsers. The resource utilization reaches nearly two times higher values.

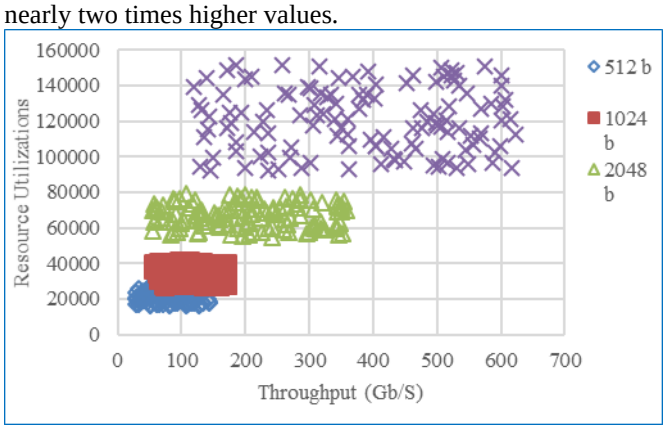

Figure 13. The relationship between Throughput and Resources of the Full parser with a different data bus width.

Figure 14 shows the latency and throughput relation of different configurations of Full parsers.

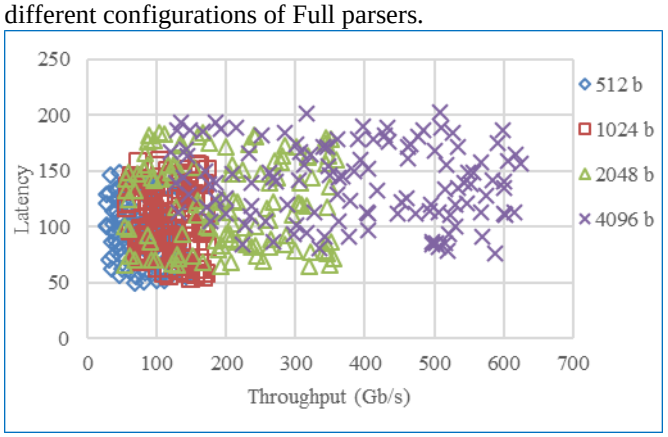

Figure 14. The relationship between Throughput and Latency of the Full parser with a different data bus width

The full parsers use more of the pipeline stages. Hence, their latency and resource utilization are much higher, nearly four times in some cases than the simple parsers. Figures 14 and 15 show the latency and throughput relation of the full parsers with different configurations.

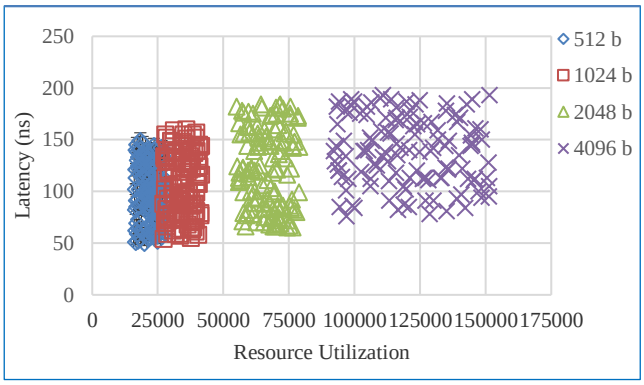

Figure 15. The relationship between Resources and Latency of the Full parser with a different data bus width

5.1.3 Summary of the Parser Test

Figure 16 shows sets of tested parsers with Pareto optimal results of the resource utilization with the throughput. The Full parsers' resource utilization is up to two times larger than the Simple parser from the graph.

Figure 17 shows Pareto optimal sets of parsers configurations in latency to throughput, where the latency

increases with the throughput.

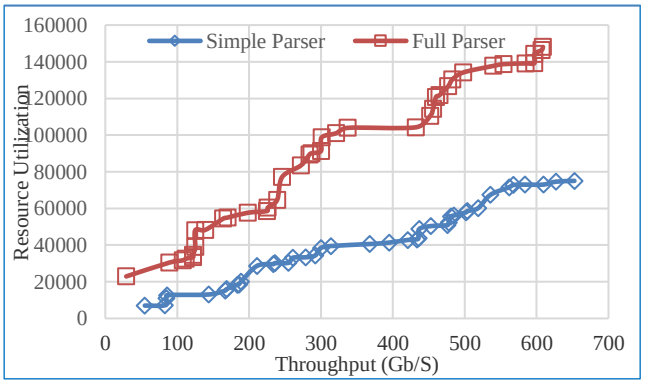

Figure 16. Pareto optimization of throughput and resources for different parses and data bus width

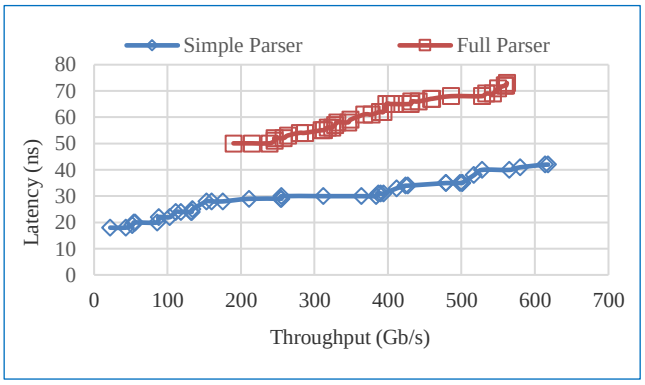

Figure 17. Pareto optimization of throughput and latency for different parses and data bus width

The results demonstrated that the data bus width increment tended to increment throughput (as intended). With doubling the data bus width, the resource utilization is slightly less than the double. The latency remained large with the increasing data bus.

5.2 Evaluation

Table 1 presents a comparative study between HP4 and other parsers: Gibb [13], handwritten VHDL parser (Golden) [6], Benácek (BK) [6] and Silva [4] with a throughput of less than or equal to 100 Gb/s in both Simple and Full parsers. HP4 nearly achieves the same performance with the same constraints while reducing the resource utilization and latency as BK [6] in the cases of Simple and Full parses. The best is the Golden parser, where it is hand-optimized. However, due to the additional pipeline registers and the full data bus width, HP4 design requires more resources than Gibb [13]. However, the HP4 throughput can reach high even in processing the shortest packets. Due to the low data bus and throughput, Gibb achieved less resource utilization than HP4.

Table 1. Comparative Study-1

Parser	Type	Data Bus [b]	Throughput [Gb/s]	Resources Utilization	Latency [ns]
Gibb [13]	Simple Parser	256	47	2.22%	N/A
Golden [6]		512	100	1.49%	15
BK [6]		512	100	3.58%	29
Silva [4]		320	100	2.03%	19.2
HP4		4096	100	2.45%	22.3
Gibb [13]	Full Parser	64	11	1.18%	N/A
Golden [6]		512	100	1.94%	27
BK [6]		512	100	3.79%	46.1
Silva [4]		320	100	2.67%	25.6
HP4		4096	100	2.98%	33.53

Also, the HP4 compared with the other parsers: Puš [2], Attig and Brebner (AB) [12], Golden parser [6], and Jakub Config [9] for the same protocol stack (Simple and Full) and throughput ranges from 100 to 400 Gb/s based on Xilinx Virtex-7 XCVH870T as shown in Table 2. The throughput of AB parser is up to 578 Gb/s, BK parser up to 478 Gb/s, and Config parser with a maximum of 926 Gb/s. HP4 (highlighted) requires less resource utilization and latency as opposed to the AB parser. The Golden and Config parsers are better in both dimensions than HP4. To overcome these limitations, HP4 can use and repeat the Golden parser (e.g., 4 x 400 Gb/s) and replace the Viretex-7 platform with UltraScale FPGA. Thus HP4 resource utilization and latency will decrease, and the throughput increase.

Table 2. Comparative Study-2

Parser	Throughput	Resources Utilization	Latency
Puš [2]	> 100Gb/s	1.19%	21.1
AB [12]		9.5%	320 ns
Golden [6]		1.94%	45 ns
Config [9]		2.05%	69 ns
HP4		5.85%	76 ns
Puš [2]	>400 Gb/s	4.88%	35.8
AB [12]		22.7%	365 ns
Golden [6]		5.87%	56 ns
Config [9]		6.38%	67 ns
HP4		7.42%	88 ns

6. Conclusions

Most of the network devices need the packet parser to complete their functions. This paper presented a novel architecture of high-performance programmable pipeline packet parser HP4 based on FPGA at the line rate.

HP4 used the P4 language to describe the packet header parsing to generate FPGA-appropriate VHDL code dynamically.

The generating unit converts the abstract definition to FPGA synthesis without in-depth knowledge of the hardware definition language. It minimized the design and implementation time. The data bus width and the number of pipeline stages are configurable parameters to optimize resource utilization, throughput, and latency.

HP4 evaluated on a variety of characteristics. The results highlight the scalability of HP4 by illustrating a wide variety of packet throughputs via adjusting the variable parameters. HP4 equilibrates the throughput, the used resources, and latency even in the worst case when parsing a set of short packets. It can parse a different line-rate throughput from 1 to over 600 Gb/s on a single Xilinx Virtex-7 870HT FPGA by considering latency and used resources in the case of the full protocol as well. Latency was about 88 ns, and resource utilization was nearly 7.42% of the FPGA's resources. Thus, the rest of the FPGA resources remain available for other requirements. HP4 can use in accelerators, smart NICs, and various network security applications (filtering and packet inspection).

References

[1] Z. Cao, H. Zhang, J. Li, M. Wen and C. Zhang, "A fast approach for generating efficient parsers on FPGAs", *Symmetry*, vol. 11, no. 10, pp. 1265, Oct. 2019.

[2] V. Pus, L. Kekely, and J. Korenek, “Design methodology of configurable high performance packet parser for fpga,” in Design and Diagnostics of Electronic Circuits & Systems, 17th International Symposium on, pp. 189–194, April 2014.

[3] X. Wang, L. Qinrang, and Y. Binghao, "A Design of Programmable Parser Generation System Based on Dynamic Programming." In 2017 International Conference on Information, Communication and Engineering (ICICE), pp. 325-328. IEEE, 2017.

[4] J. Santiago da Silva, F.-R. Boyer, and J. Langlois, "P4-Compatible High-Level Synthesis of Low Latency 100 Gb/s Streaming Packet Parsers in FPGAs," in Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 147-152, : ACM, 2018.

[5] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4 → NetFPGA workflow for line-rate packet processing," in Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, pp. 1–9, 2019.

[6] P. Bencek, V. Pu, and H. Kubtov, "P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers," in 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 148–155, May 2016.

[7] Lixin, M., Qingrang, L., & Xin, W. (), "Software-Defined Protocol Independent Parser based on FPGA", In Proceedings of the International Conference on Industrial Control Network and System Engineering Research, pp. 42-46, ACM, March 2019.

[8] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in Proceedings of the Symposium on SDN Research, ser. SOSR '17, , pp. 122–135, 2017.

[9] J. Cabal, P. Bena´cek, L. Kekely, M. Kekely, V. Pu ˇ s, and J. Ko ˇ ˇrenek, "Configurable fpga packet parser for terabit networks with guaranteed wire-speed throughput," in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, pp. 249–258, 2018.

[10] C. Kozanitis, J. Huber, S. Singh, and G. Varghese, Leaping multiple headers in a single bound: Wire-speed parsing using the Kangaroo system," in IEEE INFOCOM, pp. 830-838, 2010.

[11] Y. H. E. Yang and V. K. Prasanna, "High Throughput and Large Capacity Pipelined Dynamic Search Tree on FPGA," in Proceedings of the 18th annual ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA), pp. 83–92, 2010.

[12] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single fpga," in In Proceedings of the 2011 ACMJIEEE Seventh Symposium on Architectures for Networking and Communications Systems, ANCS '11. IEEE Computer Society, pp. 12–23, 2011.

[13] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown, "Design principles for packet parsers," in ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ser. ANCS'13. IEEE, pp. 13–24, 2013.

[14] M. Budiu and C. Dodd, "The p416 programming language," ACM SIGOPS Operating Systems Review, vol. 51, no. 1, pp. 5–14, 2017.

[15] M. Sone, " Physical Layer Security for Wireless Networks Based on Coset Convolutional Coding ", IJCNIS, vol. 12, no. 1, pp. 95-100, 2020.

[16] https://www.xilinx.com/products/design-tools/ise-design-suite.html Last Access Date 11/15/2020