Design Principles for Packet Deparsers on FPGAs

Thomas Luinaud
thomas.luinaud@polymtl.ca
Polytechnique Montréal
Montréal, Canada

Jeferson Santiago da Silva
jeda@kaloom.com
Kaloom Inc.
Montréal, Canada

J.M. Pierre Langlois
pierre.langlois@polymtl.ca
Polytechnique Montréal
Montréal, Canada

Yvon Savaria
yvon.savaria@polymtl.ca
Polytechnique Montréal
Montréal, Canada

ABSTRACT

The P4 language has drastically changed the networking field as it allows to quickly describe and implement new networking applications. Although a large variety of applications can be described with the P4 language, current programmable switch architectures impose significant constraints on P4 programs. To address this shortcoming, FPGAs have been explored as potential targets for P4 applications. P4 applications are described using three abstractions: a packet parser, match-action tables, and a packet deparser, which reassembles the output packet with the result of the match-action tables. While implementations of packet parsers and match-action tables on FPGAs have been widely covered in the literature, no general design principles have been presented for the packet deparser. Indeed, implementing a high-speed and efficient deparser on FPGAs remains an open issue because it requires a large amount of interconnections and the architecture must be tailored to a P4 program. As a result, in several works where a P4 application is implemented on FPGAs, the deparser consumes a significant proportion of chip resources. Hence, in this paper, we address this issue by presenting design principles for efficient and high-speed deparsers on FPGAs. As an artifact, we introduce a tool that generates an efficient vendor-agnostic deparser architecture from a P4 program. Our design has been validated and simulated with a cocotb-based framework. The resulting architecture is implemented on Xilinx Ultrascale+ FPGAs and supports a throughput of more than 200 Gbps while reducing resource usage by almost 10× compared to other solutions.

CCS CONCEPTS

• **Hardware → Reconfigurable logic applications**; • **Computer systems organization → Reconfigurable computing**; **High-level language architectures**; • **Networks** → *Programming interfaces.*

FPGA '21, February 28-March 2, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8218-2/21/02…$15.00
https://doi.org/10.1145/3431920.3439303

KEYWORDS

Packet deparsers; Graph optimization; P4 language

ACM Reference Format:
Thomas Luinaud, Jeferson Santiago da Silva, J.M. Pierre Langlois, and Yvon Savaria. 2021. Design Principles for Packet Deparsers on FPGAs . In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '21), February 28-March 2, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3431920.3439303

1 INTRODUCTION

The P4 Domain Specific Language (DSL) [6] has reshaped the networking domain as it allows describing custom packet forwarding applications with much flexibility. There has been a growing interest in using FPGAs to offload networking tasks. For instance, Microsoft already deploy FPGAs in their data centers to implement the data plane of Azure servers [8]. In-network computing is another avenue where FPGAs have recently been considered [20]. In addition, several recent works exploit FPGA reconfigurability to create programmable data planes and implement P4 applications [5, 13, 21].

As presented in Figure 1, a P4 application comprises three abstractions: the packet parser, the processing stage (match-action tables), and the packet deparser (§2.2). While designs of efficient packet parsers on FPGA have been widely explored [3, 5, 19], little effort has been dedicated to the implementation of efficient packet deparsers. First, to the best of our knowledge, only a single paper covers this topic [7]. However, Cabal et al. [7] report only the FPGA resource consumption for a 100 Gbps packet deparser, while the design principles and microarchitectural details are not covered. Second, as Luinaud et al. [16] have observed, a packet deparser can consume more than 80% of the resources needed to implement a complete pipeline, which can jeopardize the ability of FPGAs to implement more complex P4 applications.

This paper introduces an open-source solution to generate efficient and high-speed packet deparsers on FPGAs. It lays the foundations for the design principles of packet deparsers on FPGAs. It comprises an architecture and a compiler to generate a deparser from a P4 program.

The deparser compiler (§4) is described in Python and generates synthesizable VHDL code for the proposed deparser architecture. The generated architecture leverages the inherent configurability of FPGAs to avoid hardware constructs that cannot be efficiently implemented on FPGAs, such as crossbars or barrel shifters [1, 25].

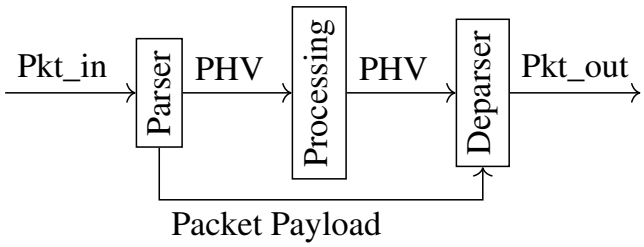

Figure 1: Considered switch architecture

The simulation environment is based on *cocotb* [11], which allows using several off-the-shelf Python packages, such as Scapy, to generate test cases. In addition, it is possible to connect the design under test with virtual network interfaces [12]. As a result, behavioural validation can be done using the P4 behavioural reference model [17].

The generated architecture has been evaluated for a variety of packet headers. The evaluations show that the generated deparser supports more than 200 Gbps packet throughput while reducing the resource usage by more than 10× compared to state-of-the-art solutions (§5).

The contributions of this paper are as follows:

- A deparser architecture that leverages FPGA configurability (§3);
- An open-source P4-to-VHDL packet deparser compiler (§4); and,
- A simulation environment based on *cocotb* to simplify deparser verification (§5).

2 PACKET PROCESSING

This section introduces the P4 language and how P4 program components are organized to describe packet processing.

2.1 P4 language

P4 [6] is an imperative DSL used to describe custom packet processing on programmable data planes.

2.1.1 Overview of P4 programs. There are four components that structure a P4 program: `header`, `parser`, `control`, and `switch`. A `header` is a structure composed of fields of specific width and a *validity bit*. A struct of headers is used to define the set of headers that can be processed by a P4 program. The `parser` block expresses in which order and how to extract packet headers. The `Control` block describes the operations to perform on headers.

2.1.2 Control Operations. In a control block, multiple operation types can be performed to modify headers. Two specific operations are of interest for the deparser, `setValid` and `setInvalid`, which can be used to set a header validity bit to valid or invalid, respectively.

In P4, control blocks also implement the deparsing logic. These blocks are composed of a series of `emit` statements. First, the order of these statements determines in which order headers are emitted. Second, a header is only emitted when its *validity bit* is set.

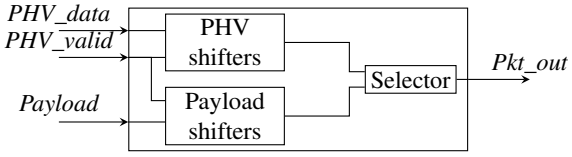

Figure 2: Deparser architecture overview

Because the sequence of `emit` statements determines the header emission order, and because the *validity bit* can be altered by previous control blocks, the deparser must be able to insert or remove headers at runtime.

2.2 P4 Program Components

This paper considers the switch structure proposed by Benáček et al. [5], composed of three parts: a parser, a processing part and a deparser, as presented in Figure 1.

Parser. The parser takes as input a packet and generates a Packet Header Vector (PHV) and a packet payload. In our design, we assume that the PHV is composed of two parts : the *PHV_data* bus containing header data and a bitmap vector, the *PHV_valid* bus, indicating the *validity bit* of each header component. We also assume that the packet payload is sent through a streaming bus with the first byte at position 0.

Processing. The processing part takes as input the PHV from the parser and outputs a modified PHV, which is forwarded to the deparser. The operations on the PHV can either be header data modifications or header *validity bit* alteration.

Deparser. The deparser block takes as input the PHV from the processing part and the payload from the parser. It outputs the packet to be sent on a streaming bus.

3 DEPARSER ARCHITECTURE PRINCIPLES

In this section we cover the deparser architecture principles. First, we introduce a deparser abstract machine. Second, we cover the deparser I/O signals. Third, we present the microarchitecture of the proposed deparser. All our design choices use the inherent configurability of the FPGAs and provide configurable blocks for the deparser compiler.

3.1 Deparser Abstract Machine

The deparser abstract machine is described in Figure 2. In this architecture, we assume that the PHV is buffered and arrives at the deparser together with a PHV valid vector.

Algorithm 1 presents the pseudo-code for the PHV shifters module while Algorithm 2 illustrates the payload shifter.

The main limiting factors to implement a deparser on FPGAs are the high number of interconnections and barrel shifters required for header insertion. To limit the use of these blocks, we construct a new packet based on the header and the payload. Thus, as the P4 deparsing logic can entirely be inferred at compile time and since FPGAs are reconfigurable, we tailor the deparser architecture to a given P4 program in order to alleviate those limiting factors.

We now cover the deparser inputs and outputs.

Algorithm 1: PHV shifter

```
input  : phv: The PHV
input  : phv_valid: PHV valid vector
output : phv_aligned_bus: The PHV aligned bus
pos ← 0;
foreach valid in phv_valid do
    if ph.isValid() then
        phv_aligned_bus.insert(pos, phv.at(pos, pos +
          ph.size())) ;
        pos ← pos + ph.size();
    end
end
return phv_aligned_bus
```

Algorithm 2: Payload shifter

```
input  : payload
input  : phv_valid: PHV valid vector
output : payload_aligned_bus: The payload aligned bus
payload_aligned_bus.insert(sum_sizes(phv_valid),
  payload) ;
return payload_aligned_bus
```

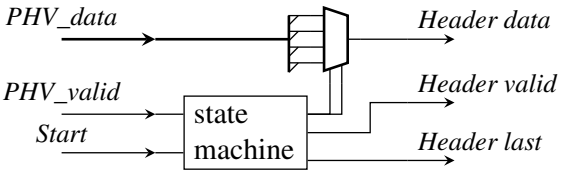

Figure 3: Header shifter for 1 bit

3.2 Inputs and Outputs

The deparser has three inputs and one output. The output *Pkt_out* and the input *Payload* are AXI4-stream buses [2]. The data width of those two buses is a compile time parameter. The two inputs *PHV_data* and *PHV_valid* respectively contain the headers data and validity bits to deparse. The width of *PHV_data* and *PHV_valid* are determined when compiling the P4 application.

3.3 Microarchitectural Details

Internally, the deparser is built around three blocks: `PHV shifters`, `Payload shifters`, and `selector`. The `PHV shifters` gets *PHV_data* and *PHV_valid* as input, and outputs a frame of headers to emit. The `Payload shifters` receives *Payload* and *PHV_valid* as input, and generates payload data frames. Both, `Payload shifters` and `PHV shifters` are inputs to the `selector`. The `Selector` generates the *Pkt_out* frames according to the status received by `Payload shifters` and `PHV shifters`.

3.3.1 The PHV Shifter. The `PHV shifters` shifts the PHV bits to build the packet. It is composed of header shifters presented in Figure 3. The maximum number of header shifters is equal to *Pkt_out* bus width.

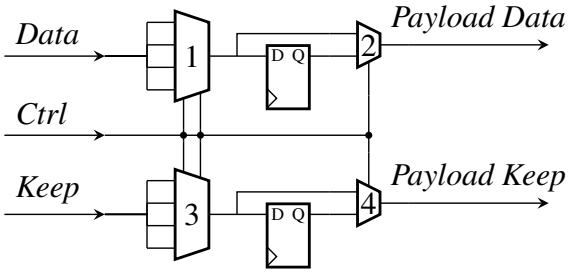

Figure 4: Payload shifter for 1 bit of data

The header shifter has three inputs: *PHV_data*, *PHV_valid* and a *Start* signal. It outputs: the *header data*, the *header valid*, and the *header last*. The *PHV_valid* and *Start* inputs are connected to the *state machine* module. The *state machine* module drives the *header valid* and *header last* output. The *PHV_data* input is connected to a multiplexer that drives the *header data* output. The multiplexer selects one of the bits of *PHV_data* based on one output of the *state machine* module. The *state machine* is derived from the deparser graph (§4), as well as the number of inputs for the multiplexer.

3.3.2 The Payload Shifter. The `Payload shifters` aligns the payload with the emitted headers. The basic block of the `Payload shifters` is shown in Figure 4. It takes *Data*, *Ctrl*, and *Keep* as inputs, and outputs *Payload Data* and *Payload Keep* signals. The bus *Data* and *Keep* are respectively connected to the AXI `tdata` and `tkeep` signals of the deparser's *Payload* input buses. Each byte of this bus is connected to one input of multiplexer 1 in Figure 4. Each bit of the *Keep* signal is connected to one input of the multiplexer 3. The *Ctrl* signal determines which input of multiplexers 1 and 3 should be selected. Finally, the output of multiplexers 1 and 3 can be registered to delay the data output by one cycle. Multiplexers 2 and 4 select either the current value or the delayed one. The value of the *Ctrl* bus is set by a small and constant associative memory generated at compile time.

3.3.3 The Selector. This block selects the right output data between `Payload shifters` and `PHV shifters` and generates the AXI4-stream output signals *Packet data*, *Packet keep* and *Packet last*. The selector takes as input the output of the `Payload shifters` and the output of the `PHV shifters` as shown in Figure 5.

The *Packet data* and *Packet keep* signals are assigned by the block *Data Select*. This block is duplicated to assign all the packet data bits. The *Packet last* signal is either *PHV last* or *Payload last* according to the presence of a payload indicated by the input signal *Has payload*.

3.3.4 Multiplexers on FPGAs. The different presented building blocks are highly dependent on the multiplexer implementation on FPGAs. We chose to use multiplexers since they are efficiently implemented on FPGAs. Indeed, a 16:1 multiplexer consumes a single slice on a Xilinx FPGA [23]. While having a very large multiplexer can become expensive, we know that the number of inputs for each multiplexer will be reduced to a minimum by the compiler (§4.2.2).

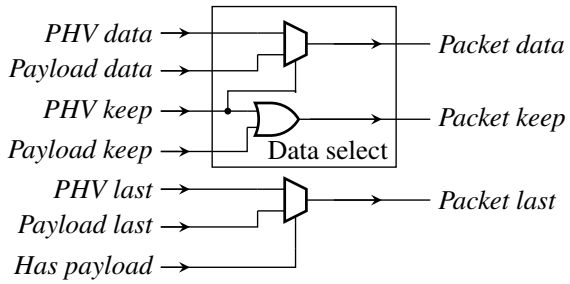

Figure 5: Deparser selector for 1 bit of data

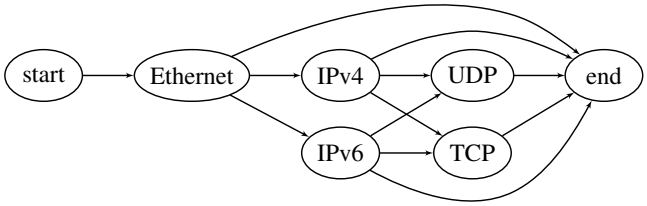

Figure 6: A possible deparser DAG with Ethernet, IPv4, IPv6, TCP and UDP headers

Table 1: Possible sequences of headers based on Figure 6 with total header size (a) and the size of each headers (b)

(a)

Path	Size (Bytes)
Ethernet	14
Ethernet->IPv4	34
Ethernet->IPv4->TCP	54
Ethernet->IPv4->UDP	42
Ethernet->IPv6	54
Ethernet->IPv6->TCP	74
Ethernet->IPv6->UDP	62

(b)

Header	Size (Bytes)
Ethernet	14
IPv4	20
IPv6	40
TCP	20
UDP	8

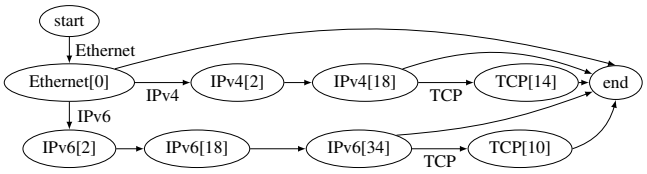

Figure 7: Sub-DAG for byte 0 from Figure 6 DAG

4 DEPARSER GENERATION

The deparser can be represented as a Directed Acyclic Graph (DAG). To generate a deparser DAG, a P4 program is compiled into a JSON file using the `p4c-bm2-ss` compiler [18]. The generated JSON file is then used to generate a deparser DAG. It is possible to optimize this DAG, but this optimization was left to future works. The rest of this section presents the process of generating the different deparser modules from a deparser DAG.

4.1 The Deparser DAG

An example of a deparser DAG, for Ethernet, IPv4, IPv6, TCP and UDP packets, is presented in Figure 6. Each node of the DAG, excluding the *start* and the *end*, represents a header. Each arrow of the DAG indicates the possible next headers to emit. Each path between *start* and *end* represents a possible set of headers to emit. The list of all possible paths is given in Table 1.

There are two parts in order to obtain a deparser from the DAG. The first part transform the deparser graph to generate the `PHV shifters`. The second part uses the deparser graph to generate the `Payload shifters`.

4.2 PHV Shifters Generation

To generate the header shifters of the `PHV shifters`, we build sub-DAGs of the deparser graph. Each sub-DAG represents one header shifter block (§3.3.1). Since most network protocols are byte aligned [5], we build one sub-DAG per output byte. This allows merging the PHV shifter state machines, hence, reduce the deparser architectural complexity.

4.2.1 Sub-DAGs Generation. In a sub-DAG, every node contains the header and the byte to extract. Each edge indicates the header validity condition to go to the corresponding next node. We propose Algorithm 3 to generate sub-DAGs.

The proposed algorithm goes through all the possible sequences of header emissions by traversing the deparser graph. We assign each byte of a sequence to a sub-DAG. When it is the first time a sub-DAG processes a byte from a specific header, we set the edge condition to this header. Figure 7 shows a sub-DAG generated with Algorithm 3 using the deparser DAG of Figure 6 and a 128-bit output bus.

4.2.2 Sub-DAG Translation. A sub-DAG translation is decomposed in two parts: the header shifter mux generation and the state machine generation. The number of inputs of the generated mux is equal to the number of nodes in the sub-DAG. The state machine is derived from the sub-DAG where each node represents a state and each edge a transition. The header byte position is converted into an input position of the multiplexer.

4.3 Payload Shifter Associative Memory Creation

The payload shifter architecture is presented in (§3.3.2). We use the graph to generate the associative memory that drives the *Ctrl* signal. This memory is generated in two steps using the deparser graph. First, we determine the set PH of possible valid headers by looking at all the possible paths between *start* and *end* in the deparser graph. Each element ph in the set PH is composed of the *PHV_valid* bus value and the sum ph_w of all headers widths.

Once the set PH is built, for each possible element ph $\in$ PH, we assign a value to *Ctrl*. For each possible *PHV_valid* value, we calculate the offset for the payload. The offset is obtained using the equation Offset = $ph_w \pmod w$, where w is the bus width and

Algorithm 3: Deparser Sub-DAGs generation

```
input  : A deparser DAG G_d, w number of sub-DAG
output : w sub graph
Sg ← list of w empty-DAG;
foreach path p between start and end in G_d do
    Sgi ← 0 ;                          // SubGraph indice
    prev_node ← [] ;
    for i ← 0 to w do prev_node.append(None);
    foreach header h in p do
        for i ← 0 to nbByte(h) do
            newNode ← node extract h position i ;
            Sg_Sgi.insert(newNode) ;
            newEdge ← edge prev_node → newNode ;
            if Sgi < i then            // conditional edge
                newEdge.set_transition_condition(h) ;
            end
            Sg_Sgi.insert(newEdge) ;
            prev_node[Sgi] ← newNode ;
            Sgi ← (Sgi + 1) mod w // next sub-DAG
        end
    end
end
return Sg
```

PH_w represents the total number of bits of the emitted header. Finally, we set the values of the *Ctrl* bit connected to the multiplexers 2 and 4 of Figure 4 for each payload shifter positioned below the offset value.

5 RESULTS

This section presents the results of this work. First, we describe the experimental setup. Then we present the impact of the compiler parameters on the generated architecture. Finally, we present and compare the implementation results with previous work.

5.1 Experimental Setup

We have generated deparsers for the following three protocol stacks:

- **T1** : Ethernet, IPv4/IPv6, TCP/UDP
- **T2** : Ethernet, IPv4/IPv6, TCP/UDP, ICMP/ICMPv6
- **T3** : Ethernet, 2 × VLAN, 2 × MPLS, IPv4/IPv6, TCP/UDP, ICMP/ICMPv6

To validate our work, we developed a simulation platform based on the *cocotb* framework [11]. We developed *cocotb* drivers and monitors for the AXI4-stream bus, allowing us to rapidly evaluate different deparser configurations. Xilinx Vivado 2019.1 was used for synthesis and place-and-route. To allow reproducibility, our codes are open[1].

5.2 Impact of Compiler Parameters

To evaluate the impact of the graph complexity, we generated and synthesized deparsers from both non-optimized deparser DAGs and parser DAGs considered as optimized deparser DAGs. Using the parser DAG as a deparser DAG was the proposed implementation in $P4_{14}$ [9]. The Block RAM (BRAM), Look Up Table (LUT), and Flip Flop (FF) usage for each synthesis run when targeting a Xilinx xcvu3p-3 FPGA are presented in Table 2. The results indicate that two factors dominate resource usage: graph complexity and data bus width.

Graph complexity. The graph complexity is impacted by the deparser code and the extent to which the graph was simplified. Since simplified deparser DAGs have fewer edges, the state machine for their PHV_shifter and the size of the associative memory for the payload shifter are reduced. In addition, fewer nodes are required for each sub-DAG. As a result, the number of inputs for the PHV shifter mux is reduced. For example, in **T1**, there are 5 headers. This results into a total of 32 paths for a non-optimized deparser DAG, while the simplified parser graph contains only 7 paths.

Bus Width. In addition to the graph complexity, the bus width impacts resource consumption. The proposed design has a latency of 6 clock cycles. Also, the latency to output a packet is a function of the total header length to emit and the bus width. As presented in Table 2, for wider buses, the worst-case latency is reduced compared to smaller buses. The worst-case latency for header emission can be calculated with the following equation:

$$\text{latency} = \left\lceil \frac{\text{total header length}}{\text{bus width}} \right\rceil + 6 \tag{1}$$

Also, increasing the bus width increases LUTs and FFs usage. For data buses varying from 64 to 256 bits, there is a slight increase in resource usage, however, this increase becomes significant for a 512-bit bus. Two factors can explain this higher complexity. First, the minimum number of multiplexer increases at a rate of 1 multiplexer per output bit. Second, with larger buses, due to header alignment, more headers can be appended to the bus for each output frame. Hence, there is less reuse of possible inputs for the PHV shifter.

5.3 Implementation Results

We also implemented non-optimized deparser DAGs for the three protocol stacks with a data bus of 512 bits. We compared the deparser implementation results against the deparser generated by the Xilinx SDNet 2017.4 [24] and Benáček et al. [5] deparsers. The results of these implementations are shown in Table 3. Our deparser supports a throughput 20 Gpbs greater than the deparser proposed by Benáček et al. [5], while reducing by 5× the resource usage. Compared to the parser generated by Xilinx SDNet [24], in the worst case, our deparser supports a throughput that is 60 Gbps lower, but our deparser uses almost 10× fewer resources.

When comparing implementation and synthesis results, the resource consumption remains stable. In the case of **T1** and **T2**, the performance after place-and-route is almost the same. However, the maximum clock frequency could be improved by pipelining the multiplexers, without significantly impacting resource consumption. Indeed, the generated architecture consumes less than one FF per slice while a typical slice posses eight FFs. As a result, unused FFs could be used to pipeline multiplexers, as they would be unlikely to be driven by other modules.

[1] https://github.com/luinaudt/deparser/tree/FPGA_paper

Table 2: Synthesis results of deparser DAG on a Xilinx xcvu3p FPGA

Test	width (bits)	worst latency (cycles)	Deparser DAG				Parser DAG			
			LUTs	FFs	BRAMs	Frequency	LUTs	FFs	BRAMs	Frequency
T1	64	19	1517	402	0	448 MHz	630	395	0	448 MHz
	128	13	2066	784	0	448 MHz	1019	782	0	448 MHz
	256	10	2862	1522	0	448 MHz	1722	1365	0	448 MHz
	512	8	9127	3002	0	469 MHz	5777	2696	0	469 MHz
T2	64	20	1798	404	0	448 MHz	746	402	0	448 MHz
	128	13	2664	808	0	448 MHz	1075	758	0	448 MHz
	256	10	4879	1602	0	469 MHz	1667	1370	0	469 MHz
	512	8	11212	3197	0	448 MHz	5811	2691	0	448 MHz
T3	64	22	2137	390	6	448 MHz	1039	372	2	448 MHz
	128	14	3962	780	14	448 MHz	1858	758	5	448 MHz
	256	10	6598	1525	32	448 MHz	3842	1450	29	448 MHz
	512	8	14287	3116	41	469 MHz	8603	2931	32	469 MHz

Table 3: Implementation results for a 512 bits output data bus compared with previous work

Test	Work	Slice	LUT	FF	BRAM	Throughput (Gbps)
T1	Our	3144	9 k	3 k	0	200
	SDNet	N/A	77 k	95 k	116.5	160
	SDNet	N/A	78 k	95 k	116.5	256
T2	Our	3922	11.2 k	3.2 k	0	220
	SDNet	N/A	98 k	119 k	149.5	240
	[5]	20 k	N/A	N/A	N/A	120
T3	Our	4770	14 k	3 k	20.5	140
	SDNet	N/A	137 k	165 k	229.5	160
	SDNet	N/A	139 k	165 k	229.5	220
	[5]	24 k	N/A	N/A	N/A	120

6 RELATED WORK

Wang et al. proposed P4FPGA [22]. P4FPGA is an open-source and vendor-agnostic P4-to-FPGA compiler targeting mid-performance FPGAs (10 Gbps). Ibanez et al. [14] proposed integrating Xilinx SDNet P4 compiler [24] to the off-the-shelf NetFPGA board [15]. This work exposes the SDNet limitation in implementing the deparser logic, which turned out to be the module with the largest resource consumption of the generated pipeline [16]. Indeed, as per our experiments, we observed that Xilinx SDNet was unable to optimize unreachable paths in the deparsing graph.

Benáček et al. [5] presented automatic generation of P4-based packet parsers and deparsers to VHDL. This work extends to deparsers their previous research on packet parsers [4]. However, the deparser architecture and design principles have not been covered because the optimizations were derived from the design of packet parsers. Other packet parsers research include [3, 10, 19]. Gibb et al. [10] introduced general design principles for packet parsers, but does not cover the case of packet deparsers. In addition, Attig and Brebner [3] proposed a language to represent parsers with an architecture and a compiler to implement them on an FPGA. Also, Santiago da Silva et al. [19] proposed to use graph optimizations, similarly to our work, to simplify the parser pipeline.

7 CONCLUSION

P4 has changed the networking landscape as it allows expressing custom packet processing. In recent years, several works have mapped P4 programs to FPGAs. However, the majority of these works have focused on implementing packet parsers or match-action stages. To date, no general packet deparsing principles on FPGAs have been proposed. Indeed, the naive approach of previous works on generation of deparsing logic has made hardware implementation of this block very costly on FPGAs. In this work, we tackle this problem by introducing a set of design principles for implementing packet deparsers on FPGAs. In our work, we proposed an architecture tightly coupled to the FPGA microarchitecture in order to leverage the FPGA's inherent configurability. We also demonstrate the importance of deparser graph simplification to reduce resource consumption. Our results show that our proposed deparser architecture crosses the 100 Gbps throughput boundary while reducing the resource consumption by one order of magnitude. Finally, to permit reproducibility, we open-sourced our framework and an integrated simulation environment based on *cocotb*.

ACKNOWLEDGEMENTS

The authors thank Thibaut Stimpfling and Bochra Boughzala from Kaloom for their insightful comments. This work was supported by Kaloom, Intel, Noviflow, the Natural Sciences and Engineering Research Council of Canada, and Prompt.

REFERENCES

[1] M. S. Abdelfattah and V. Betz. 2012. Design tradeoffs for hard and soft FPGA-based Networks-on-Chip. In *2012 International Conference on Field-Programmable Technology*. 95–103.
[2] AMBA. 2010. *Axi4-stream protocol specification.* Technical Report. ARM®.
[3] Michael Attig and Gordon Brebner. 2011. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems (ANCS '11)*. IEEE

Computer Society, Washington, DC, USA, 12–23. https://doi.org/10.1109/ANCS.2011.12

[4] P. Benácek, V. Pu, and H. Kubátová. 2016. P4-to-VHDL: Automatic Generation of 100 Gbps Packet Parsers. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 148–155.

[5] Pavel Benáček, Viktor Puš, Hana Kubátová, and Tomáš Čejka. 2018. P4-To-VHDL: Automatic generation of high-speed input and output network blocks. *Microprocessors and Microsystems* 56 (2018), 22 – 33. https://doi.org/10.1016/j.micpro.2017.10.012

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[7] J. Cabal, P. Benacek, J. Foltova, and J. Holub. 2019. Scalable P4 Deparser for Speeds Over 100 Gbps. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 323–323. https://doi.org/10.1109/FCCM.2019.00064

[8] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[9] P4 Language Consortium. 2017. *The P4 Language Specification, version 1.0.4*. Technical Report. https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf

[10] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. 2013. Design principles for packet parsers. In *Architectures for Networking and Communications Systems*. 13–24.

[11] Chris Higgs and Stuart Hodgeson. 2020. *cocotb*. https://github.com/cocotb/cocotb

[12] Chris Higgs and Stuart Hodgeson. 2020. *Tutorial: Ping — cocotb 1.4.0 documentation*. https://docs.cocotb.org/en/stable/ping_tun_tap.html

[13] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3289602.3293924

[14] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. 2019. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/3289602.3293924

[15] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. 2007. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. 160–161.

[16] T. Luinaud, T. Stimpfling, J. S. da Silva, Y. Savaria, and J. M. P. Langlois. 2020. Bridging the Gap: FPGAs as Programmable Switches. In *2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR)*. 1–7. https://doi.org/10.1109/HPSR48589.2020.9098978

[17] p4Lang. 2020. *behavioral-model: The reference P4 software switch*. https://github.com/p4lang/behavioral-model

[18] p4lang. 2020. *p4c: $P4_{16}$ reference compiler*. https://github.com/p4lang/p4c

[19] Jeferson Santiago da Silva, François-Raymond Boyer, and J.M. Pierre Langlois. 2018. 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 (FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 147–152. https://doi.org/10.1145/3174243.3174270

[20] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 21, 16 pages. https://doi.org/10.1145/3302424.3303979

[21] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3050220.3050234

[22] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research (SOSR '17)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3050220.3050234

[23] Xilinx®. 2016. *7 Series FPGAs Configurable Logic Block*. Technical Report. https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[24] Xilinx®. 2018. SDnet Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdnet.html

[25] Peter Yiannacouras, Jonathan Rose, and J. Gregory Steffan. 2005. The Microarchitecture of FPGA-Based Soft Processors. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '05)*. Association for Computing Machinery, New York, NY, USA, 202–212. https://doi.org/10.1145/1086297.1086325