An Explicitly Parallel Architecture for Packet Processing in Software Defined Networks

An Explicitly Parallel Architecture for Packet Processing in Software Defined Networks

Hesam Zolfaghari
Electrical Engineering Unit
Tampere University
Tampere, Finland
hesam.zolfaghari@tuni.fi

Davide Rossi
Department of Electrical, Electronic, and Information Engineering
University of Bologna
Bologna, Italy
davide.rossi@unibo.it

Jari Nurmi
Electrical Engineering Unit
Tampere University
Tampere, Finland
jari.nurmi@tuni.fi

***Abstract*— Programmable data plane is a key enabler of Software Defined Networking. By making networking devices programmable, novel networking services and functions could be realized by means of software running on these devices. In this paper, we present a lightweight packet processor that could process the packets on the fly as they arrive. As we will see, the area of this packet processor is smaller than a packet parser employing Ternary Content Addressable Memory. As an added benefit, the designed packet processor could also reduce the traffic to the lookup tables on the chip. Moreover, its use is not limited to switches and routers. It could also be used in the Network Interface Cards and offload packet processing tasks. Despite its packet processing capabilities, packet processor instances required for sustaining aggregate throughput of 640 Gbps have area equivalent to the packet parser instances in the Reconfigurable Match Tables Architecture.**

Keywords*— *On-the-fly packet processing, programmable data plane, Explicit Parallelism, Packet Processor

I. INTRODUCTION

Networking gear has traditionally been fixed-function and tied to a specific set of protocols. Recently, the benefits of programmable data plane have forced the industry and academia to design programmable networking devices. Commercially available programmable packet processing systems are Barefoot Tofino [1], Intel FlexPipe [2] and Netronome Agilio [3]. Benefits of programmability in the data plane are as follows:

- Support for newer networking protocols
- Simpler networking devices
- Telemetry and network troubleshooting
- Offloading computation to the network

With programmable data plane, support of different network protocols is made possible by means of software updates. The architectures presented in [4], [5] and [6] are all programmable. All it takes for a device to provide new functionality is to run the required software. Although it sounds counter-intuitive, programmable networking devices are architecturally simpler considering the range of functionality that they could provide. If this range of functionality were to be provided using conventional design principles, a large amount of protocol-specific state must be permanently stored.

One of the strongest driving forces of deploying programmable data plane is the rich set of telemetry and network troubleshooting facilities. In [7], [8] and [9] network state is embedded into the packets and later on used for diagnostics and telemetry. With the data plane being programmable, any internal state of the device which is of significance to the network could be written to the packet and later on used for analysis. In fact a network administrator-defined header could be inserted into the packet to carry such state. Finally, the latest trend in programmable data plane is in-network computing. In this paradigm, computational tasks are offloaded by networking devices. For instance, in [10], the programmable network device is used as a neural network accelerator.

The packet parsing and packet processing subsystems are the main components of networking devices such as switches and routers. Earlier, we designed a programmable packet parser in [11] and [12]. In this programmable packet parser, the output format is identical to that of the packet parser in [1], [4] and [5]. The internal architecture of our parser, however, is fundamentally different. Rather than a state machine that relies on a Ternary Content Addressable Memory (TCAM) to derive the next state, our parser uses a program control unit. In this paper, we have augmented the parser with functional units for packet processing. As a result, it is a Packet Processor and we refer to it as such in this paper. As we will see, for an aggregate throughput of 640 Gbps, the total area of our Packet Processor instances equals that of the packet parsers in Reconfigurable Match Tables (RMT) architecture while providing wide range of packet processing capabilities in addition to packet parsing.

This paper is organized as follows. In section II the motivation for this architecture is elaborated. In section III the architecture is presented followed by a packet processing example in section IV. Finally, the evaluation results will be presented in section V.

II. MOTIVATION

The set of packet processing operations that must be performed on a packet can be categorized under two classes:

- Standardized operations
- Deployment-specific operations

Standardized operations are the ones that must be performed on a packet because the protocol standard instructs to do so. For instance, in IPv6, a packet's Hop Limit must be examined and if its value is zero, the packet must be discarded. The packet parser could do this as the packet is arriving. However, the treatment for a packet that has a given Destination Address is deployment specific. For instance, it could lead to the packet being discarded or the packet being forwarded to all egress ports. The packet parser in its conventional form could not be involved with this level of packet processing unless it has a lookup table at its use, in which case it could do almost any deployment-specific packet processing.

Our packet parser reads the header of the incoming packet in units of maximum 4 bytes wide. Throughout the time of a packet's arrival at which point the packet parser is involved, many packet processing tasks could be performed. For

instance, validity checking of a packet and field modifications that do not depend on the outcome of table lookups could be already done. By doing so, by the time the packet has fully arrived, packet processing has been partially done. This greatly reduces the time required for processing of packets, as the packets spend less time in the packet processing device. Processing of header fields during the course of parsing comes at a negligible cost.

In addition, the parsers in the RMT architecture have their output multiplexed into a serial pipeline. This means that if new packets are constantly arriving through the ingress ports, the output of the parser has to wait until its turn for entering the pipeline comes. With 16 parser instances and one packet processing pipeline, this waiting could last as long as 16 cycles during which a lot could be done. By starting packet processing already at the time the packet starts arriving, this waiting time is exploited for packet processing.

In order to enhance the chance of being able to perform all the required processing for a packet, the concept of packet flows is fully exploited. The parser is equipped with tiny binary lookup tables that contain the lookup result for the most recently arrived packets. The lookup result is retrieved from the main processing pipeline which contains lookup tables. If a match is found in the tiny lookup tables, the corresponding action that could not have otherwise been possible could be performed in the Packet Processor. In this case the packet could bypass the main packet processing pipeline. Even if a match is not found, the packet enters the main pipeline in partially processed form which means that it requires less processing time.

III. Architecture

The packet processing system comprises a programmable packet parser and a programmable packet processor. The two entities work in harmony for on-the-fly packet processing. The architecture of the programmable packet parser is discussed in detail in [12]. It extracts the header fields of the incoming packets and writes them to a storage location called Original Header Word (OHW) entries. These entries are marked R16-R31. The programmable Packet Processor does not wait for the packet parser to fully parse a packet before it starts processing of the packet. It starts as soon as an entry is written to the OHW. Fig. 1 illustrates a high-level view of the Packet Parser and the Packet Processor.

The packet processor is comprised of 4 Groups of Functional Units (GFU). Each GFU has 8 functional units which are sufficient for processing a number of header fields. Together, these GFUs perform all the processing required for the headers of an arriving packet. Fig. 2 illustrates a GFU. As we can see, each GFU contains two ALUs, two shifters, a Merge Unit, a Load and Store unit, a Lookup Table and a Parameters Unit. The Load and Store Unit, Lookup Table and Parameter Unit are stateful units. For instance, it is possible to store the number of packets associated with a specific search key in the memory located in the Load and Store Unit. All the other units are stateless units.

The complete instruction set contains 64 instructions. Depending on their semantics and the functional unit which executes them, there are 8 classes of instructions. Instruction classes have been outlined in TABLE I.

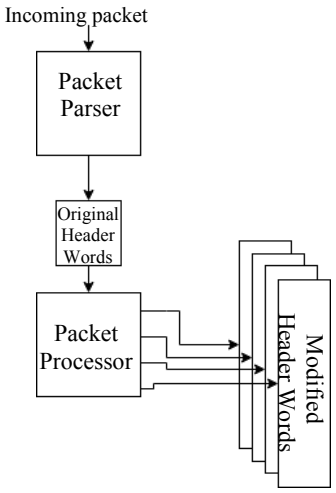

Fig. 1. High-level view of the Packet Parser and the Packet Processor

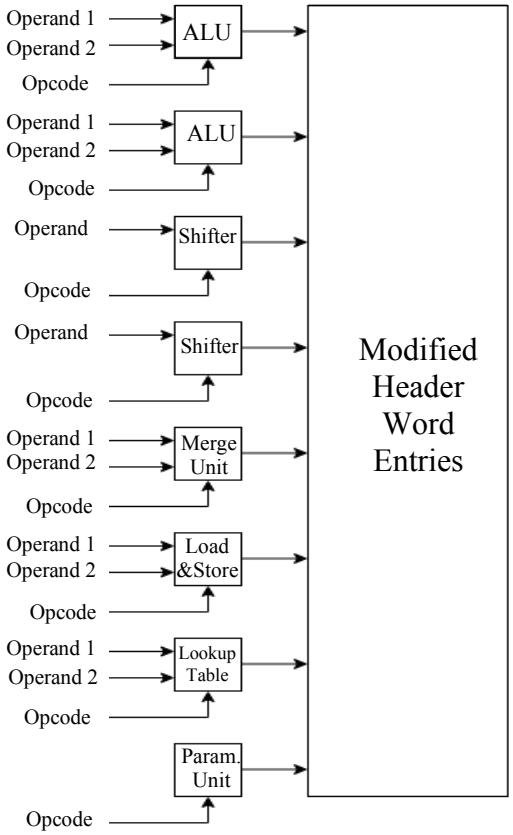

Fig. 2. A Group of Functional Units (GFU)

Merge Word instruction is used to merge the corresponding bytes of a 32-bit word into a target word in the specified manner. For each byte in the target word, there is a choice of two from the corresponding byte position of the two source words. It is used for forming search keys and finalizing the words of a header. Save Search Key in conjunction with Load and Store instructions allow to save a key generated during processing of a packet and associate data with it so that they could be later on retrieved if the same search key is encountered. Bit Extract instruction is useful for reading flag bits such as Explicit Congestion Notification (ECN) bits. Load Parameter is used for inserting four different types of values:

TABLE I. INSTRUCTION SET

Instruction Class	Operations	Mnemonic
Condition Evaluation	Check for equality, greater than or less than conditions	CEQ, CG, CL
Arithmetic and Logic	Arithmetic and Logic operations	ADD, SUB, AND, OR, NOT, MOVE, XOR, NAND
Shift	Logical left and right in 1-, 4-, 8- and 16-bit units	SHL1, SHL4, SHL8, SHL16, SHR1, SHR4, SHR8, SHR16
Merge	Merge the corresponding bytes of two words	MERGE0000, MERGE0001, …, MERGE1111
Lookup	Search key lookup and lookup table maintenance	Save Search Key, Lookup
Bit Extraction	Extract the specified bit within a byte	BE0, BE1, … , BE7
Load and Store	Load and Store operations	Load, Store
Load Parameter	Insert header template, processing operand, status or field value	Load Param(i)

- Header templates such as header words for Internet Control Message Protocol (ICMP)
- Unique values for fields intended to contain unique values such as a connection identifier
- System status such as status of queues
- Operands for functional units

The functional units receive up to two operands from four different sources:

- Modified Header Word (MHW) entries in the GFU (R0-R15)
- Original Header Word (OHW) entries (R16-R31)
- Certain entries of Modified Header Word in other GFUs (R32-R63)
- An immediate value

There is one OHW space which is shared by all GFUs. All operands from the OHW entries are subject to extraction as a result of which an 8- or 16-bit field within the 32-bit word is extracted and zero-extended to 32 bits prior to execution. Alternatively, the selected 32-bit OHW entry could be operated upon without any field extraction.

In addition to the operands, the functional units receive a one-bit predicate input from the 16-bit condition status (r0-r15) to which the result of condition evaluation instructions and lookup result is written. Each GFU has an independent condition status word. The predicate input is used for conditional execution. Fig. 3 illustrates the inputs to the ALU.

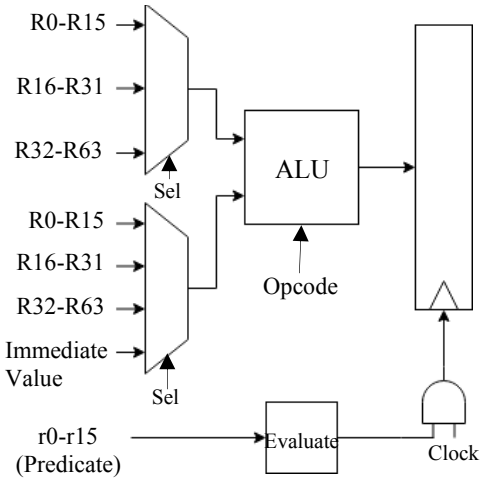

Fig. 3. Inputs to the ALU

There is an MHW space per GFU. MHW storage is a multiport register file in which each functional unit has its own writing space. This means that no other functional unit could write to the range of locations reserved for the functional unit in question. Two entries are reserved per functional unit within the GFU. In contrast, any functional unit could read from the writing space of both itself and all other functional units. If the nature of a given header allows, each functional unit produces a result that is read on the next cycle by another functional unit and this process goes on until the header is fully processed. This is an ideal scenario and also requires that the software written for the program also makes use of the functional units properly. The datapath of each GFU contains logic for operand forwarding. Thus, the functional units need not wait for the input to be written to the MHW if another functional unit has produced the value in the preceding clock cycle. Independent GFUs could also communicate, but it requires a number of cycles from the time a functional unit in a GFU has produced the result until another functional unit in another GFU could read that value because there is no cross-GFU forwarding. OHW entries contain arriving header words written by the parser.

A. Data Types

All data types are 32 bits wide. The parsing subsystem could read from a buffer of incoming packets in 8-, 16- and 32-bit units but writes them to 32-bit entries within the OHW storage. Therefore, everything that the packet processing subsystem reads from this space is 32 bits wide.

B. Memories

There are four different memory blocks in each GFU:

- Memory for holding the search key entries in the binary lookup table
- Memory for holding one-time-usable values
- Memory for holding parameter values
- Memory for load and store operations

C. Program Control

There are three key mechanisms for ensuring correct program flow.

- Branches based on header field values
- Branches based on results produced by the Packet Processor
- Conditional execution performed by the functional units of the Packet Processor

The functionality for both branches mentioned above is provided by the program control unit of the packet parser. The purpose of branches based on results produced by the functional units of the Packet Processor is to avoid a long stream of instructions whose execution result is not written to the MHW entries. Such streams of instructions waste cycles and prevent functional units from being used for other instructions.

D. Packet Processing Metadata

The metadata is a data structure containing information regarding the packet under processing. The rest of the system uses the metadata of a packet for further decisions regarding a packet. Metadata fields are outlined in Table II.

TABLE II. METADATA FIELDS

Metadata Field	Width (bits)	Purpose
Incoming Port	6	Indicates the port through which the packet has arrived.
Processing Status	2	Indicates whether processing is done and also whether the packet requires further processing in the main pipeline.
Discard	1	Specifies if the packet should be discarded.
Destination	2	Specifies if the packet should be sent to: -Main processing pipeline -Outgoing port -Control Plane -Internal Buffer
Destination Port Bitmap	64	Specifies the outgoing port(s) through which the packet must be transmitted.

IV. ILLUSTRATED EXAMPLE

In this section, we illustrate how this architecture processes IPv4 packets. We have specifically chosen IPv4 because it constitutes a great portion of Internet traffic. In addition, it requires quite a lot of processing even in the absence of header options. We will demonstrate how this programmable architecture can handle this workload without any protocol-specific hardware.

At a minimum, once an IPv4 packet arrives at a router, the checksum must be verified to detect possible errors during transmission. The Time to Live (TTL) field must be evaluated to see if the packet can continue its path or if it must be discarded. If it can continue, the value of TTL must be decremented and as a result of this, the checksum must be recalculated. The Destination Address field of the header must be used as a key to lookup into the forwarding table to determine the outgoing port(s) for the packet. In this illustrated example, we assume that the packets carry no header options. But to increase the workload to some extent, we perform some additional integrity checking on the header. Fig. 4 contains the packet processing functions to execute on this Packet Processor.

```
1  void process_ipv4_packet(ipv4_packet *p)
2  {
3          verify_ipv4_packet(p);
4          check_TTL(p -> TTL);
5          update_checksum(p);
6          lookup_destination_address(p ->
   Destination_Address);
7          check_DF(p -> flags);
8  }
9  void verify_ipv4_packet(ipv4_packet *p)
10 {
11         verify_version(p);
12         verify_IHL(p);
13         verify_Total_Length(p);
14         verify_checksum(p);
15 }
```

Fig. 4. Source code for processing IPv4 packets

As we can see, it contains functions for verifying the contents of Version, Internet Header Length (IHL) and Total Length fields. In order to clarify the contents of these functions, Fig. 5 illustrates the contents of the function that checks the value of TTL.

```
1  void check_TTL(ipv4_packet *p)
2  {
3          if(p -> TTL == 0)
4          {
5              insert_ICMPv4_header(BAD_HEADER);
6              drop(p);
7          }
   }
```

Fig. 5. Source code for checking the TTL field

The compiler or the programmer must decide how to assign these functions to the processing resources of the system. The processing resources of this architecture are:

- GFUs
- Functional units within each GFU
- Registers

We assign all integrity checking functions other than checksum checking to GFU 0. Checksum calculation and update could share the intermediate results for more efficient execution, therefore we assign both functions to GFU 1. We assign checking of Don't Fragment (DF) flag to GFU 2 and finally GFU 3 performs address lookup. Tables III, IV, V and VI contain the instructions that will be executed at each clock cycle in GFUs 0, 1, 2 and 3 respectively. t_0 is the time at which the first header word of the incoming IPv4 header is written to the OHW storage. It should be noted that the tables contain the instructions that are executed in each of the GFUs and not the

complete source code written for processing of IPv4 packets. Furthermore, any instruction that uses a given OHW entry as operand must be scheduled for execution at least 2 cycles after the packet parser has written it to the corresponding OHW entry. This is to ensure that the operand fetch logic has retrieved the correct value.

TABLE III. INSTRUCTIONS EXECUTED IN GFU 0

Time	Operation
t_0	-
t_1	-
t_2	R4 <- SHR4 R16(3)
t_3	r0 <- R4 CEQ 0x04 R3 <- R16(3) AND 0x0F
t_4	(r0) R14 <- Param(15); r4 <- R3 CG 0x04; R4 <- SHL4 R3
t_5	(r4) R14 <- Param(15); r0 <- R16(4) CEQ R4; r4 <- R16(4) CG R4
t_6	r15 <- r0 OR r4
t_7	r15 <- NOT r15
t_8	(r15) R14 <- Param(15)

In GFU 0, the value of Version field is obtained to be compared with the immediate value of 4. Similarly, the value of IHL is obtained to ensure that it is greater than or equal to 5. IHL is then shifted left four bits to derive the header size in bytes. Then it is compared with the value of Total Length to ensure that the entire size of the packet is greater than or equal to the header size in bytes. If any of the above-mentioned conditions is not fulfilled, ICMP bad header is inserted.

TABLE IV. INSTRUCTIONS EXECUTED IN GFU 1

Time	Operation
t_0	-
t_1	-
t_2	R0 <- R16(4) + R16(6)
t_3	R0 <- R0 + R17(4)
t_4	R0 <- R0 + R17(6)
t_5	R0 <- R0 + R18(4); R3 <- MOVE R0
t_6	R0 <- R0 + R18(6); R3 <- R3 + R19(4)
t_7	R0 <- R0 + R19(4); R3 <- R3 + R19(6)
t_8	R0 <- R0 + R19(6); R3 <- R3 + R20(4)
t_9	R0 <- R0 + R20(4); R3 <- R3 + R20(6)
t_{10}	R0 <- R0 + R20(6); R2 <- R18(3) - 0x01
t_{11}	R4 <- SHR16 R0; R6 <- SHL16 R2
t_{12}	R0 <- R0 + R4; R6 <- SHL8 R6
t_{13}	R1 <- NOT R0; R8 <- R18(7) MERGE R6
t_{14}	r0 <- R1 CEQ 0x00; R7 <- SHR16 R8
t_{15}	(NOT r0) R14 <- Param(15); R3 <- R3 + R7
t_{16}	R6 <- SHR16 R3
t_{17}	R3 <- R3 + R6
t_{18}	R9 <- R8 MERGE R3

In GFU 1, all 16-bit words of the header are added together for verifying the checksum. These additions take 10 cycles because the addition result must be accumulated and there are at least 10 items to be added together. R0 holds the summation result. At t_5 addition result calculated up to that point is copied into R3 so that the calculation of the new checksum could begin in parallel. As we can see, during t_5-t_{11} at each cycle there are two ALU or two shift operations occurring without conflict.

At t_{10}, TTL is decremented and stored in R2. The decremented TTL value which is one byte wide is then shifted all the way to take the most significant byte position. This takes two clock cycles. Then it is merged with the third word of the header so that it contains the updated TTL. Then a copy of this updated word is shifted right 16 bits for updating the checksum. Once the updated checksum is calculated, it is merged with the updated TTL and the unchanged Protocol field at t18.

TABLE V. INSTRUCTIONS EXECUTED IN GFU 2

Time	Operation
t_0	-
t_1	R14 <- Param(15)
t_2	r4 <- R16(4) CG R14
t_3	r14 <- BE5 R17(1)
t_4	r15 <- r4 AND r14
t_5	(r15) R15 <- Param(14)

GFU 2 executes instructions for checking the DF flag. At t1, R14 is loaded with the value of maximum allowed packet size. The router has previously calculated this using Path MTU Discovery. At the following clock cycle, the value of Total Length field which contains the size of the entire packet in bytes is compared with the value of R14. At the next clock cycle, the DF flag is extracted for evaluation. At t4, the two conditions of packet size being larger than maximum allowed packet size and DF flag being set are subject to an AND operation. At t5, ICMP Destination Unreachable is inserted if the compound condition calculated in the preceding cycle turns out to be true.

TABLE VI. INSTRUCTIONS EXECUTED IN GFU 3

Time	Operation
t_0	-
t_1	-
t_2	-
t_3	-
t_4	-
t_5	-
t_6	R12 <- Lookup R20(7)
t_7	-
t_8	(r6) R10 <- Load R12

GFU 3 performs lookup operation. At t6, Destination Address field of the arrived packet is submitted to the lookup table in the GFU for lookup. It takes one cycle until a matching entry is found and it takes another cycle to derive the address of the entry which contains the associated data which in this case is the set of ports to which the packet must be sent. The associated entry is loaded at t8 if a match has been found.

V. EVALUATION

In this section we present the implementation results. The Packet Processor has been implemented in VHDL. We have synthesized it on 28 nm UTBB FD-SOI technology in worst-case operating conditions (1.0V, ss, 125°C) using Synopsys

Design Compiler J-2014.09-SP4. Power analysis was also performed in worst-case operating conditions at the supply voltage of 1.0V (ss, 125°C). We have verified that all timing constraints are met for operation at the frequency of 2.0 GHz.

Table VII contains the area of the packet processing units called atoms in [6] and the equivalent functional units of our Packet Processor. Since the atoms perform only field modification and not lookup, we have also excluded the effect of the lookup tables on the values of table VII. It should be noted that we have not implemented the atoms in [6]. Instead we have provided the area results for the aggregate of functional units which provide functionality equivalent to the atoms in [6]. The atoms in [6] were synthesized to a 32-nm standard cell library. Using (1), we convert their area to equivalent 28 nm values so that we could compare them with the results of our own implementation.

$$\text{Area in 28 nm} = \text{Area in 32 nm} \times (28/32)^2 \qquad (1)$$

TABLE VII. AREA COMPARISON OF ATOMS IN [6] AND EQUIVALENT FUNCTIONAL UNITS IN THIS WORK

Atom	Area in [6] (μm^2)	Area in this work (μm^2)	Saving (%)
Stateless	1059	700	34
ReadWrite	191	90	53
ReadAddWrite	330	245	26
Predicated ReadAddWrite	605	418	31
IfElse ReadAddWrite	753	630	16
Subtract	1164	834	28

The Stateless atom could only perform arithmetic, logic, relational and conditional operations on operands. The Read/Write atom has the smallest area among the stateful atoms. The distinctive feature of atoms with higher area is the addition of logic levels for predication and logic for performing predicate-specific actions. From the perspective of functionality, the functional units in this work are far superior to the atoms because as we can see from Table 4 and Table 5 of [6], the functional unit in the stateful atoms performs either addition or addition and subtraction. The rest of the logic is for evaluating the predicate. In our architecture, with two ALUs in each GFU, action associated with each outcome for the predicate could be done in parallel using different functional units. It should also be noted that our architecture runs at 2.0 GHz while the architecture in [6] runs at 1.0 GHz.

Table VIII contains the area results of different components of a single instance of our Packet Processor. The table presents area also in terms of number of gates so that the results could be more easily compared with the results in [4] and [5]. For each component, the equivalent gate count is obtained by dividing the total cell area by the area of the smallest cell used in the technology. It should be noted that memories have an independent read port for each Packet Processor instance and hence they could be shared without contention. In other words, they need not be replicated per Packet Processor instance. In order to be able to make a precise comparison between the area of this Packet Processor and the Packet Parser in RMT, we must answer the critical question of how many Packet Processor instances are required for sustaining the target aggregate throughput of 640 Gbps. Assuming that the packets at each port arrive on a 10 Gbps link and that the packets are all minimum-sized, a new packet will arrive every 67 nanoseconds. Equivalently, there are around 15 M packets per port per second.

TABLE VIII. AREA FIGURES FOR A SINGLE PACKET PROCESSOR INSTANCE SYNTHESIZED TO 28 NM UTBB-FDSOI

Component	Area (μm^2)	Area (gate count × 1000)
Packet parsing logic	5193	11
Packet parsing parameter memories	96593	197
Program Control Logic	342	0.7
Packet processing functional units	56612	116
Operand storage, fetch and forwarding	75896	155
Instruction memory	344650	704

We also assume that all Layer-2 functionality is performed by an interface between the link and the Packet Processor which performs IPv4 routing. As we saw in section IV, the processing takes 19 cycles which in a 2.0 GHz system is equivalent to 9.5 nanoseconds. Therefore, one Packet Processor instance per 6 ports is sufficient for sustaining the line rate. However, we assign a Packet Processor per 4 ports in case some of the packets require further processing. Assuming that these Packet Processors will be in the same port configuration as in [5], 16 Packet Processor instances are sufficient to sustain the 640 Gbps throughput. The sum of the area of 16 instances is roughly 5.4 M gates which is slightly smaller than the area of the packet parser instances in [5].

VI. CONCLUSION AND FUTURE WORK

In this paper we presented a fully programmable architecture for protocol-independent packet processing. We have seen that by enhancing the previously designed packet parser with packet processing functionality which comes at a marginal cost, a great deal of packet processing could be done on the packets even if a match is not found in the tiny lookup tables. As for future work, we would like to enhance the throughput of the system and fine-tune it for high-end workloads and applications. In addition, requirement for use of more advanced functional units with the aim of supporting as wide range of protocols while maintaining protocol-independence must be investigated.

ACKNOWLEDGMENT

We hereby express our gratitude to Mr. Glen Gibb from Barefoot Networks for the invaluable comments regarding our idea. This work has been partially supported by the Finnish DELTA doctoral training network.

REFERENCES

[1] The World's Fastest & Most Programmable Networks https://www.barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/

[2] Intel Ethernet Switch Fm6000 Series 10/40 GbE Low Latency Switching Silicon https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf

[3] Agilio FX SmartNIC https://www.netronome.com/products/agilio-fx/

[4] Gibb, Glen, George Varghese, Mark Horowitz, and Nick McKeown. "Design principles for packet parsers." In Architectures for Networking and Communications Systems, pp. 13-24. IEEE, 2013.

[5] Bosshart, Pat, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN." ACM SIGCOMM Computer Communication Review 43, no. 4 (2013): 99-110.

[6] Sivaraman, Anirudh, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. "Packet transactions: High-level programming for line-rate switches." In Proceedings of the 2016 ACM SIGCOMM Conference, pp. 15-28. ACM, 2016.

[7] Kim, Changhoon, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J. Wobker. "In-band network telemetry via programmable dataplanes." In ACM SIGCOMM. 2015.

[8] Jeyakumar, Vimalkumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. "Millions of little minions: Using packets for low latency network programming and visibility." In ACM SIGCOMM Computer Communication Review, vol. 44, no. 4, pp. 3-14. ACM, 2014.

[9] Handigol, Nikhil, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. "I know what your packet did last hop: Using packet histories to troubleshoot networks." In 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), pp. 71-85. 2014.

[10] Sanvito, Davide, Giuseppe Siracusano, and Roberto Bifulco. "Can the network be the AI accelerator?." In Proceedings of the 2018 Morning Workshop on In-Network Computing, pp. 20-25. ACM, 2018.

[11] Zolfaghari, Hesam, Davide Rossi, and Jari Nurmi. "An Explicitly Parallel Architecture for Packet Parsing in Software Defined Networks." In 2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 1-4. IEEE, 2018.

[12] Zolfaghari, Hesam, Davide Rossi, and Jari Nurmi. "Low-latency Packet Parsing in Software Defined Networks." In 2018 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), pp. 1-6. IEEE, 2018.