Reducing Crossbar Costs in the Match-Action Pipeline

Hesam Zolfaghari
Department of Electrical Engineering
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
Department of Electrical Engineering
Tampere University
Tampere, Finland
jari.nurmi@tuni.fi

Abstract— **Software Defined Networking (SDN) is a new networking paradigm in which the control plane and data plane are decoupled. Throughout the recent years, a number of architectures have emerged for protocol-independent packet processing. One such architecture is the Protocol Independent Switch Architecture (PISA). It is a programmable and protocol-independent architecture composed of a number of Match and Action stages. Inside each of these stages is a crossbar to generate the search key and another crossbar to provide the input to the Action Units. In this paper, we design and explore alternative interconnection schemes with the aim of finding the most area- and power-efficient interconnection structure. Moreover, we propose further modifications to the interconnection structure, as a result of which the on-chip area of both match and action crossbars will be reduced by more than 70 % and power dissipation will be reduced by 25.8 % and 23.1 % for match and action crossbars respectively.**

Keywords—Software Defined Networking, Protocol Independent Switch Architecture, Crossbar

I. Introduction

Packet processing is a domain in which performing matching and applying the actions associated with the match result are the key operations. The match operation could be used to determine the outgoing port for a packet or determining the kind of processing that a group of fields requires. Programmable packet processing systems must have the means to process different protocols each of which requires corresponding match and action operations. Therefore, a programmable packet processing system should be able to generate a search key using any subset of the header fields. Moreover, the functional units performing tasks other than lookup must be able to operate on any of the header fields. This is achieved by the use of flexible crossbars.

There are a number of programmable networking devices already available on the market. Intel FlexPipe [1], Cavium XPliant [2] and Barefoot Tofino [3] are the most notable of such devices. Due to the abundance of flexible tables and processing units, we limit our focus to the Barefoot Tofino. It can be programmed using the P4 language [4] for providing a wide range of packet processing functionalities. The internals of Tofino are based on the Protocol Independent Switch Architecture (PISA) which is a switch architecture first proposed in [5]. PISA is comprised of a programmable packet parser based on the architecture proposed in [6]. The programmable parser extracts header fields and places them in placeholders of three different sizes, 8-bit, 16-bit and 32-bit entries. These entries together form a 4096-bit vector of header fields. This vector is called Packet Header Vector (PHV) and it traverses a pipeline of 32 stages. Each stage contains a match part and an action part. Using a large crossbar, two 640-bit search keys are generated for table lookup in the match stage. The result of the lookup determines the required actions to be performed in the subsequent action stage. The action stage contains an action unit for each one of the entries. Therefore, there are action units of the different bit widths referred to earlier. For each action unit, there is a multiplexer which selects the inputs. The first input to the action units is from the entries of the PHV and the second input is either from the PHV or action memories containing packet processing parameters. Being a 32-stage pipeline, the crossbars make a noticeable contribution to the overall area of the chip. For instance, the area of match crossbars is 6 mm^2 in total [5].

There are Match and Action crossbars in each pipeline stage of the PISA. According to [5], each of the 1280 bits of the search key are driven by a 224-to-1 multiplexer. The multiplexers are constructed using a binary tree of AOI22 gates. The choice of 224-to-1 multiplexers indicates that there is an alignment constraint for the placement of header fields into the search key. This means that bytes can be placed into any location within the search key, 16-bit fields must be placed at even locations and 32-bit fields must be placed at locations whose index is a multiple of four. Such constraints help limiting the complexity of crossbars.

The Action crossbar in PISA provides input to the Action Engines. The first input to the Action engines is selected from the PHV while the second input is selected from either the PHV or the Action memories. The Action crossbar allows for combining smaller header fields. For instance, two 16-bit fields could be combined to form a 32-bit input to a 32-bit Action Engine.

The crossbars used in the original paper are not the only crossbars that can fulfill the interconnection requirements. In this paper we devise alternatives for providing the required interconnection, and compare area and power dissipation of each alternative. Furthermore, we justify the use of more lightweight crossbars and observe the results. The rest of the paper is organized as follows: In section II, different alternatives for implementing match and action crossbars will be evaluated. In section III, actual requirements for match and action crossbars will be analyzed. In sections IV and V, we propose smaller crossbars and present the required architectural modifications. Section VI presents the experimental results followed by the conclusion.

II. Alternative Crossbar Architectures for PISA

There are numerous strategies for forming the match key as well as selecting the inputs to the Action Engines. In this section we explore some of the alternatives with the aim of finding the most efficient solution.

The match crossbar in PISA operates at bit level, meaning that while adhering to the alignment constraint mentioned earlier, each bit of the search key is selected independently. Consequently, there are 10240 bits of select inputs required in total for the multiplexers.

A. *Alternative Match Crossbars*

1) Byte-level match field selection

Alternatively, we could select input to the search key on byte-level basis while still maintaining the alignment constraint. The PHV could be thought of as being comprised of 512 bytes and the search key to be generated as 160 bytes. Therefore, we could fill the search key using 160 512-to-1 8-bit multiplexers. In this organization, 1440 bits of select are required.

2) Word-level match field selection

Another solution is to operate at 32-bit basis using two levels of multiplexers. In the first level, there are four 64-to-1 8-bit multiplexers, two 96-to-1 16-bit multiplexers and one 64-to-1 32-bit multiplexer. Using these multiplexers, different combinations of fields could be combined to form 32 bits of the search key. For instance, two 8-bit fields and one 16-bit field could be combined together. The multiplexer at the second stage selects the combination of first-level multiplexers' outputs. Fig. 1 depicts the internals of this solution for filling in 32 bits of the 1280-bit search key. For other 32 bits of the search key, the same structure must be replicated. Some of the multiplexers could share the select lines. For generating a 1280-bit search key using this structure, 1160 bits of select are required.

We have implemented the alternatives in VHDL and synthesized them 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 performed in typical operating conditions at the supply voltage of 1.1V (tt, 25°C). The same methodology has been used throughout the whole paper. Timing analysis confirms that all crossbars mentioned in this paper can run in a 1.0 GHz system.

Table I compares the above-mentioned alternatives for the Match crossbar in terms of area and power. The given values are for one stage of the Match-Action pipeline. Moreover, their output generates a 640-bit match key. As we can see, the byte-level crossbar has the largest area and power dissipation values. In addition, they have more levels of logic compared to other crossbars. As a result, scaling the frequency beyond a point requires using registers. This increase latency and area. Using the word-level interconnection scheme results in 6 % reduction in area and 13 % reduction in power dissipation compared to the original scheme used in [5].

B. *Alternative Action Crossbars*

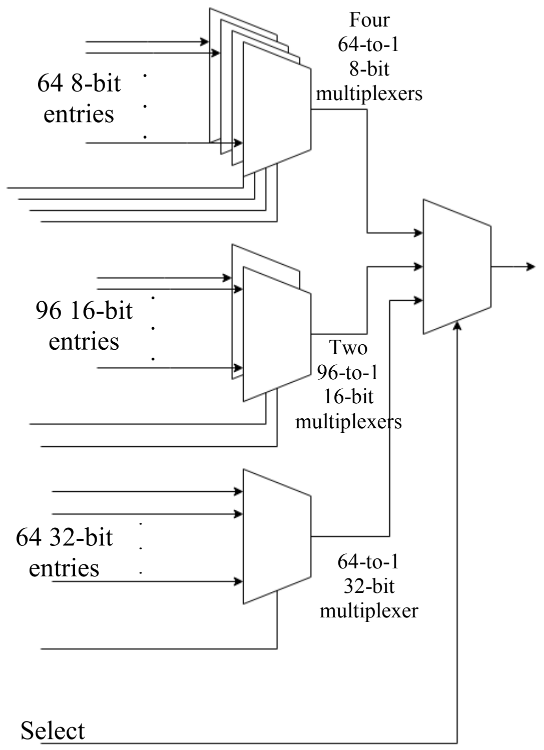
The use of interconnection schemes in which the width of select values is extremely wide should be avoided because that will increase the required width for instruction memory.

1) Bit-level selection

For the Action crossbar, if the same approach as the Match Action is adopted, 32768 select bits are required for the multiplexers.

2) Byte-level selection

If, we choose to select the input on byte-level, 512 multiplexers are required, each being a 512-to-1 8-bit multiplexer. Under this scheme, the number of select bits will be reduced to 4608 bits.



Four 64-to-1 8-bit multiplexers
64 8-bit entries
96 16-bit entries
Two 96-to-1 16-bit multiplexers
64 32-bit entries
64-to-1 32-bit multiplexer
Select

Fig. 1. Generating 32 bits of the search key using two levels of multiplexers

TABLE I. COMPARISON OF MATCH CROSSBAR ALTERNATIVES

Interconnection Strategy	Area (μm^2)	Total Power Dissipation (mW)
Bit-level	125650	137.5
Byte-level	256772	156
Word-level	118724	119.4

3) Combination of smaller processing units

In this implementation, the input to the 8-bit Action Engines comes from any of the 8-bit entries of the PHV, while the input to the 16-bit Action Engines comes either from combination of any two 8-bit entries or any of the 16-bit entries. Similarly, the input to 32-bit Action Engines is selected from any two 16-bit entries or any of the 32-bit entries of the PHV. Therefore, the input to 16-bit and 32-bit action engines comes from two levels of multiplexers. For this scheme, 3648 bits of select are required. Fig. 2 illustrates multiplexers required for an action unit of each size under this scheme.

4) Zero-extension of smaller processing units

In the final Action crossbar that we consider, 8-bit Action Engines take any of the 8-bit entries as input while 16-bit action engines receive either a 16-bit entry or a zero-extended 8-bit entry whose width is 16 bits after zero-extension. Similarly, 32-bit action engines take any 32-bit entry or any 16-bit entry which has been zero-extended to 32 bits. This scheme requires 1664 bits of select which is the smallest number among the solutions discussed so far. However, the actual merging of smaller entries requires two steps: Selecting two zero-extended values as input to a given Action Engine and using the Deposit Byte operation code for the Action Engine in question to merge the inputs together. Therefore, the Action crossbar by itself cannot perform the actual merging and requires the assistance of the action engine. However, the simplicity of this scheme makes it an interesting choice.

Table II presents area and power values for different Action crossbars. The given values are for one stage of the Match-Action pipeline. Similar to the match crossbars, byte-level crossbars are the least efficient alternative. The most important observation is that the action crossbar with zero-extension is the most efficient crossbar. Compared to the bit-level interconnection scheme, use of the two-level crossbar with zero-extension capability results in 52.7 % and 46.7 % reduction in area and power dissipation respectively.

TABLE II. COMPARISON OF ACTION CROSSBAR ALTERNATIVES

Interconnection Strategy	Area (μm^2)	Total Power Dissipation (mW)
Bit-level	804160	880
Byte-level	1643343	992.4
Combination of smaller processing units	503423	656.2
Zero-extension of smaller processing units	379992	468.7

III. REQUIRED CROSSBAR COMPLEXITY

In this section, we analyze the actual required complexity for both Match and Action crossbars.

A. *Required Match Crossbar Complexity*

Typically, a search key is comprised of header fields whose semantics allow for the concept of matching. For instance, in Internet Protocol version 4 (IPv4), the destination address field is used to lookup into the forwarding table to determine the outgoing port of the packet. As another example, the Next Header field in Internet Protocol version 6 (IPv6) is the match key for determining the kind of processing that the header following IPv6 header requires. In many processing scenarios, the search key is made up of multiple header fields placed next to each other. For instance, Protocol, source address and destination address from IPv4 header as well as source port and destination port from Transmission Control Protocol (TCP) header can be used together to form a search key for retrieving the flow-specific parameters.

As we can see, not all the header fields have the semantics to be used as a search key. Therefore, use of crossbars that receive input from the whole entries of the PHV is unnecessary. Smaller crossbars could be used without degradation in functionality.

B. *Required Action Crossbar Complexity*

Determining the right degree of complexity for Action crossbars is more complicated than that of Match crossbars. In order to do so, various packet processing actions must be taken into consideration. We have considered commonly used protocols because protocol-independent packet processors should be capable of processing them for the transition to these devices to be feasible. We are primarily interested in the number of inputs to each action because the number of inputs determines the complexity of the required crossbar. The inputs are mainly header fields. However, metadata could also be the input.

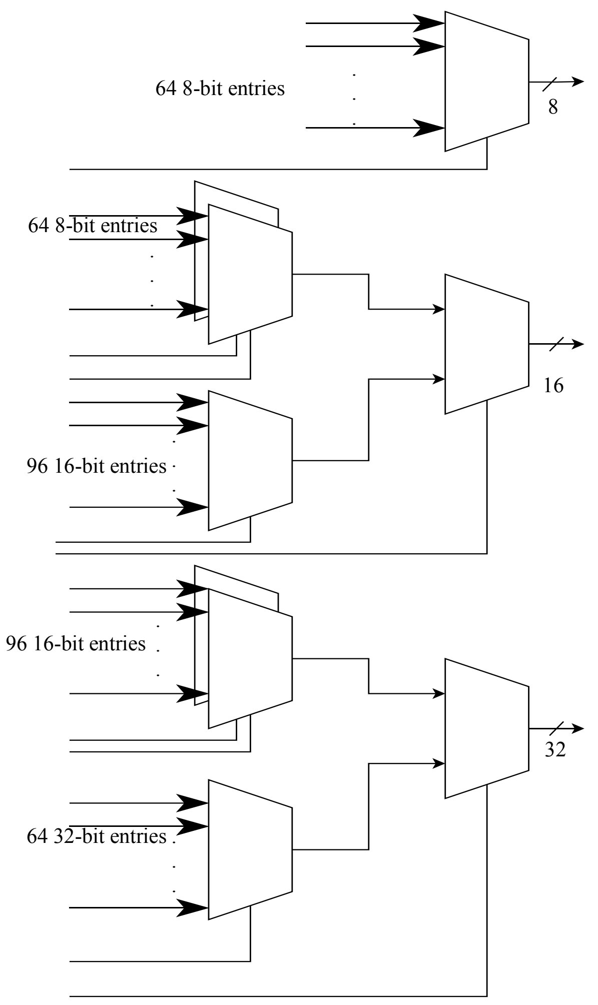


Fig. 2. Part of the Action crossbar in which smaller processing units can be combined

At one extreme, we encountered actions which require only one or two inputs. A prime example of this group of actions is decrementing the Time to Live (TTL) field in Internet Protocol version 4 (IPv4) header. Fig. 3 illustrates the corresponding subroutine.

```
void IPv4_TTL_Decrement(ipv4_packet* p)
{
        //TTL is located at byte offset 8
        if(*(p + 8) == 0)
        {
                drop(p);
        }
        else
        {
                *(p + 8) -= 1;
        }
}
```

Fig. 3. Source code for decrementing Time to Live in IPv4

At the other extreme, there are actions which require access to the entire header. In other words, they operate on all the fields existing in the header. An example of such actions is the Internet checksum [7] used in IPv4 and Internet Control Message Protocol version 4 and 6. Fig. 4 illustrates the subroutine for calculating the checksum in IPv4.

There are important observations to make from the Internet checksum calculation. It takes 16-bit words as input. Therefore, the programmable parser must place incoming header into the 16-bit entries of the PHV. Operations such as TTL decrement which operate on smaller input must copy the 16-bit word containing the TTL into smaller entries of the PHV to perform the desired operation. Alternatively, if the programmable parser places the header fields into entries that best match the fields' actual size, they need to be combined into 16-bit words for Internet checksum calculation. Being one's complement sum, the additions must be performed by 32-bit Action Units. Therefore, all the 16-bit words must be copied to 32-bit entries. Once all the required additions are performed, the lower 16-bit portion of the result will be copied to a 16-bit entry.

```
void calculate_IPv4_checksum(ipv4_packet* p)
{
        unsigned int sum = 0;
        unsigned char IHL= *(p + 0) & 15;
        unsigned int temp;
        for(int i = 0; i <= IHL * 4; i = i + 2)
        {
             //Adding 16-bit words
             temp = (unsigned int)(*(p + i) * 256)
+ (*(p + i + 1));
             sum += temp;
        }

        //as long as there is overflow
         while(sum >= 65536)
        {
             sum = (sum & 65535) + (sum>>16);
        }
        sum = ~sum;
        //disposing the value of the upper 16 bits
        sum = sum & 65535;
}
```

Fig. 4. Source code for calculating IPv4 checksum

Between the two extremes mentioned above, there are actions which require a handful of header fields. For instance, in order to determine which queue a packet must be sent to in an IPv6 router with Quality of Service (QoS) support, Traffic Class, Flow Label and outgoing port are required.

Based on this discussion, we can come up with the following taxonomy of header fields:

- Fields as they appear in the technical specification of a networking protocol: For instance, the Version field is a 4 bit field in the header of Internet Protocol version 6.
- Fields in the form required for packet processing actions: As an example, Source and Destination Addresses are 32-bit fields in Internet Protocol version 4, but in order to calculate checksum, they must be broken into 16-bit fields.
- Fields as placed in the entries of the PHV by the programmable parser and crossbar: The PHV in the PISA architecture contains entries of three sizes. Header fields whose size does not exactly match these entries have to share placeholder with other fields or span over more than one placeholder. Examples of such header fields are the 4-bit Internet Header Length field in Internet Protocol version 4 and 20-bit label field in Multiprotocol Label Switching.

IV. Reducing Crossbar Costs

As we saw in the previous section, only a fraction of fields existing in a header are used for forming match keys. In addition, not all packet processing actions require all the header fields. Based on this observation, the packet parser and crossbars in the Match-Action pipeline can be modified with the following goals:

- Tailoring the header fields to the packet processing actions: Part of the time required for processing a packet is contributed to by transferring them to a suitable PHV entry. If we could limit this part of the processing, the overall packet processing time will be reduced. This specifically concerns the action crossbar in which smaller processing units are zero-extended. Using these crossbars, the actual merging of smaller processing units takes another cycle.
- Reducing the complexity of crossbars by limiting the number of inputs: The crossbars in PISA allow for full interconnection, meaning that all entries are the input to each of the multiplexers. Except for tasks such as checksum calculation, this is rarely encountered.

Based on the arguments above, we would like to limit the number of PHV entries that can be selected as the input to its corresponding Action unit. In other words, we can think of the PHV as being divided into independent segments. Inside each segment, the input to each entry's Action unit can be any other entry within the segment. However, only few of the entries of other segments can be accessed. The programmable parser extracts incoming header fields and places them into entries of the PHV segments. In order to limit the communication of independent segments, it can place a given header field into more than one segment, meaning that duplicates can exist among the segments. Moreover, it can extract an incoming 32-bit header such that it fills two 8-bit and one 16-bit entries in one PHV segment and at the same time it fills two 16-bit entries of another PHV segment. By doing so, the fields will be tailored to packet processing tasks. It should be pointed out that the programmable parser does this according to the software that is written for it. As such, it does not have any built-in knowledge of protocols.

We have chosen the value of 4 as the number of independent segments. Alternatively, we could think that the number of inputs to the multiplexers is divided by four. This means that inside each segment, there will be 16 8-bit entries, 24 16-bit entries and 16 32-bit entries. In addition, the match crossbar receives as input only a quarter of the entries present in the whole PHV. This is equivalent to the number of entries in one segment. However, the match crossbar receives an even number of inputs from each of the four segments.

V. Architectural Requirements

The first item that must be modified is the packet parser. For each independent PHV segment, it must have a separate output port. We use the packet parser proposed in [8] and [9]. It has a program control unit as opposed to a Ternary Content Addressable Memory (TCAM). As a result, its area is considerably lower than that of the parser used in [3], [5] and [6]. Reduced area leaves room for further modifications. A high-level view of this parser is illustrated in Fig. 5. As we can see, there are four 8-bit, two 16-bit and one 32-bit output port. At any given instance in time, only a fraction of these ports has valid data. Data at these ports will be written to the PHV. The PHV is organized as a number of banks. Therefore, each of the aforementioned ports can write to its corresponding bank once it has valid data. The only required modification is replication of the PHV Filler component inside the parser so that incoming header fields can be extracted independently. This means increase in the width of the instruction. The instruction field associated with the PHV Filler is 4 bits wide. In the new architecture, 16 bits will be required for this purpose. In addition, the location of extracted fields within the PHV is specified by software. 20 bits are required to specify the location of extracted fields. In the modified parser, 48 bits are required in total for specifying location of extracted fields in the PHV.

Another required modification is the resizing and arrangement of banks that together comprise the PHV. The width of the PHV is still 4 Kilobits. Only the banks comprising the PHV will be resized.

The most performance-critical aspect is interconnection of independent segments. We must ensure that limiting the number of inputs to crossbars has as little effect as possible on the packet processing capabilities and throughput of the architecture. In order to maximize the savings in area and power dissipation, we consider the action interconnection scheme in which zero-extended values of smaller processing units will be the input to the next larger multiplexer. Under this organization, in each segment, there will be 16 8-bit 16-to-1, 24 16-bit 64-to-1 and 16 32-bit 64-to-1 multiplexers. Both 16-bit and 32-bit 64-to-1 multiplexers have 24 inputs that could come from the other segments because for both of them the total number of inputs from their own segment is 40. These 24 input lines could be used to pass the value of entries in other segments. With 24 16-bit multiplexers each having 24 inputs that could come from other segments, there is room for 576 16-bit entries which is far more than the total number of 16-bit entries in the entire PHV. Similarly, with 16 32-bit multiplexers each having 24 unused inputs, 384 32-bit inputs can be accommodated which is well above the total number of 32-bit entries of the PHV. Each multiplexer with unused inputs can take an equal number of entries of matching size from the other segments. For instance, each of the 64-to-1 16-bit multiplexers in a given segment could take 8 entries from each of the other segments to fill its 24 unused inputs. Alternatively, one such multiplexer could take all the 16-bit entries of another segment with its adjacent multiplexer taking all the 16-bit entries of another segment as its inputs.

In addition, because the number of action engines matches the number of PHV entries, each entry has an action unit reserved for it and hence, it is not required to take the same entry as the second input to the action unit in question. Consequently, each multiplexer inside a segment can take one entry from the other segments. As each segment contains 56 entries, each segment can in total take 56 entries belonging to other segments. Therefore, independent segments can also communicate reasonably well. The only limitation is that when an entry residing in another segment is required, only certain action engines have access to it. However, once the value of an entry residing in another segment is written to an entry in the current segment, then all action units of the current segment have access to it.

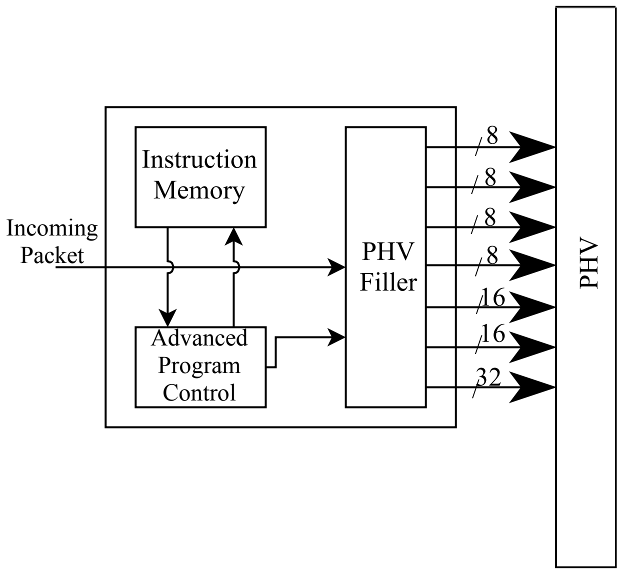


Fig. 5. High-level view of the packet parser

The deparser does not require any modification because the overall size of the PHV remains the same. Moreover, in the original architecture too, there will be PHV entries that contain only intermediate results that must be discarded.

VI. Experimental Results

In this section, we present the costs and savings of modifications. Table III compares area and power of the two parser variants. As we can see, the increase in area is only 3.4%. The increase in switching power is due to presence of more registers in the modified parser. However, as the parser accounts for less than 1 % of the chip area [5], this increase can be neglected. The savings reflected in the upcoming tables outperform this increase. Tables IV and V present area and power results of lightweight match and action crossbars respectively. For each one of the crossbar variations discussed in section II, the figures for the corresponding lightweight crossbar is provided. It should be noted that values in table V reflect area and power results of the crossbars in all four segments within a Match-Action stage, not only one segment. The area and power saving columns specify how much area and power has been saved for each crossbar with respect to its corresponding larger instance. Since for all variants, the number of inputs to the multiplexers in the crossbars is divided by four, the savings in area are all in the same range of 70 %. The saving in power dissipation is at least 25 % for the match crossbars and at least 23 % for the action crossbars. The ratio of crossbar areas is maintained. For instance, the word-level match crossbar is still the most lightweight crossbar. It is also worth mentioning that using lighter crossbars, the frequency could be scaled up more easily because there will be fewer levels of logic.

TABLE III. COMPARISON OF PARSER IMPLEMENTATIONS

	Baseline Parser	Parser with multiple outputs
Total Area (μm^2)	29900	30937
Internal Power (mW)	7.7	8.2
Switching Power (mW)	4.2	11.7
Leakage Power (mW)	4.5	4.8
Total Power (mW)	16.4	24.7

TABLE IV. AREA AND POWER DISSIPATION COMPARISON OF LIGHTWEIGHT MATCH CROSSBARS

Match Crossbar Variation	Area (μm^2)	Saving in area (%)	Total Power Dissipation (mW)	Saving in power dissipation (%)
Bit-level	34885	72.2	93.5	32
Byte-level	67499	73.7	101.8	34.7
Word-level	32813	72.3	88.5	25.8

TABLE V. AREA AND POWER DISSIPATION COMPARISON OF LIGHTWEIGHT ACTION CROSSBARS

Action crossbar variation	Area (μm^2)	Saving in Area (%)	Total Power Dissipation (mW)	Saving in power dissipation (%)
Bit-level	223264	72.2	598.4	32
Byte-level	431996	73.7	651.6	34.3
Combination of smaller processing units	140260	72.1	503.2	23.3
Zero-extension of smaller processing units	107000	71.8	360.3	23.1

VII. CONCLUSION

In this paper, we devised alternative interconnection schemes while maintaining the interconnection requirements. Based on the experimental results, we determined the most area- and power-efficient crossbars for the Match-Action pipeline. In addition, by analyzing headers and packet processing actions, we proposed use of smaller crossbars using which area will be decreased by at least 70 %. The savings in power dissipation are between 23 and 34 %. By careful placement of header fields in PHV entries, any possible performance degradation will be eliminated. Moreover, by making the compiler aware of cross-segment interconnection restrictions within a Match-Action stage, the changes proposed in this paper will be invisible to the programmer.

ACKNOWLEDGMENT

We would like to acknowledge the Finnish DELTA network as well as The Pekka Ahonen Fund for providing the partial funding for this project.

REFERENCES

[1] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[2] Cavium's XPliant™ Ethernet Switch Supports the Emerging Open Ecosystems https://www.cavium.com/Caviums-XPliant-Ethernet-Switch-Supports-The-Emerging-Open-Ecosystems.html

[3] Barefoot Networks, "The world's fastest and most programmable networks," [Online]. Available: https://barefootnetworks.com/resources/worlds-fastest-mostprogrammable- networks

[4] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G. & Walker, D. (2014). P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review, 44(3), 87-95.

[5] Bosshart, P., Gibb, G., Kim, H. S., Varghese, G., McKeown, N., Izzard, M., Mujica, F. & Horowitz, M. (2013, August). Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In ACM SIGCOMM Computer Communication Review (Vol. 43, No. 4, pp. 99-110). ACM.

[6] Gibb, G., Varghese, G., Horowitz, M., & McKeown, N. (2013, October). Design principles for packet parsers. In Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on (pp. 13-24). IEEE.

[7] Rijsinghani, A. (1994). Computation of the internet checksum via incremental update.

[8] Zolfaghari, H., Rossi, D., & Nurmi, J. (2018, July). 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.

[9] Zolfaghari, H., Rossi, D., & Nurmi, J. (2018, October). 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.