BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

HABILITATION THESIS

Brno, 2018

Ing. Jan Kořenek, Ph.D.

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

HARDWARE ACCELERATION IN COMPUTER NETWORKS

HARDWAROVÁ AKCELEARCE V POČÍTAČOVÝCH SÍTÍCH

HABILITATION THESIS
HABILITAČNÍ PRÁCE

AUTHOR **Ing. JAN KOŘENEK, Ph.D.**
AUTOR PRÁCE

BRNO 2018

Abstract

The speed of network traffic processing is a crucial parameter for most devices and systems, because any packet drop can cause lower quality of network services, affect precise monitoring or disallow detection of security threats. General purpose processors are not able to process all data on high-speed network links. For 100 Gbps links, every packet has to be processed in less than 5 ns. Network devices widely use hardware acceleration to speed up time-critical operations. Therefore, this thesis deals with five widely used time-critical operations together with corresponding hardware architectures, which are able to achieve wire-speed 10, 40 or even 100 Gbps throughput. In particular, the thesis deals with packet parsing and header fields extraction, longest prefix matching (IP look-up), packet classification, pattern matching and deep packet inspection. All these operations are widely used in precise network monitoring systems, network security devices and also in the infrastructure of data centres. The thesis introduces how deep pipelines, perfect hashing and pipelined automata can help to achieve 100 Gbps throughput and decrease hardware resources. A novel concept is introduced to accelerate deep packet inspection. The concept provides software flexibility together with high performance, because fast packet processing is controlled at the level of flows by software modules. Moreover, proposed concept and hardware architectures are used in hardware accelerated network security and monitoring devices, which has been transferred to successful commercial products and used to monitor and protect CESNET2 academic network.

Keywords

Hardware, acceleration, networks, monitoring, security, Ethernet, FPGA, card.

Abstrakt

Rychlost zpracování síťové provozu je pro většinu síťových zařízení klíčovým parametrem, neboť při velkém objemu dat a nízké výkonnosti zařízení může docházet ke ztrátám paketů. Ztráta paketu se pak sekundárně může projevit nižší kvalitou poskytovaných služeb, nižší přesností monitorování sítě nebo může zamezit detekci bezpečnostních hrozeb. Současné procesory nejsou dostatečně výkonné pro zpracování síťového provozu na dnešních vysokorychlostních síťových linkách. Pro zpracování síťového provozu na rychlosti 100 Gb/s musí být každý paket zpracován za méně než 5 ns. Aby bylo možné zajistit zpracování paketu v takto krátkém čase, využívají síťová zařízení k urychlení časově kritických operací hardwarovou akceleraci. Práce se proto zabývá hardwarovou akceleraci pěti nejčastěji používanými časově kritickými operacemi a představuje hardwarové architektury, které jsou schopny zajisti pro tyto operace zpracování síťového provozu na plné rychlosti 100 Gb/s. Konkrétně se jedná o operace pro analýzu a extrakci položek z hlaviček paketů, operaci hledání nejdelšího společného prefixu (prefixů IP adres), klasifikaci paketů, hledání řetězců a o analýzu na úrovni aplikačních protokolů. Všechny tyto operace se používají v systémech pro přesné monitorování sítí, bezpečnostních zařízeních, ale například i v infrastruktuře datových center. Práce ukazuje jak je možné s využitím hlubokých zřetězených linek, perfektních hash funkcí a zřetězených automatů dosáhnout plné propustnosti 100 Gb/s a snížit nároky na hardwarové zdroje. Současně je představen nový koncept pro analýzu síťového provozu na úrovni aplikačních protokolů. Navržený koncept poskytuje kromě vysoké výkonnosti flexibilitu na úrovni softwarového zpracování, neboť hardwarová akcelerace je řízena na úrovni síťových toků pomocí softwarových modulů. Navržený koncept a hardwarové architektury byly již použity pro urychlení monitorovacích a bezpečnostních nástrojů, byly integrovány do několika komerčně úspěšných zařízení a jsou využity i pro ochranu a monitorování akademické sítě CESNET2.

Klíčová slova

Hardware, akcelerace, sítě, monitorování, bezpečnost, Ethernet, FPGA, card.

Reference

KOŘENEK, Jan. *Hardware Acceleration in Computer Networks*. Brno, 2018. Habilitation thesis. Brno University of Technology, Faculty of Information Technology.

Hardware Acceleration in Computer Networks

Declaration

Hereby I declare that this habilitation's thesis was prepared as an original author's work. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Jan Kořenek
March 18, 2018

Acknowledgements

I would like to thank Prof. Ing. Lukáš Sekanina, Ph.D. for his professional help and advices in making this work. I would also like to thank all people from Cesnet for their valuable comments, especialy to Viktor Puš, Lukáš Kekely and Jiří Matoušek.

Contents

Chapter 1

Introduction

First computer networks are dated to 1960, when the US Department of Defence started ARPANET project to build robust, fault-tolerant communication networks. The ARPANET project has been evolved to current Internet in 1990s, when ARPANET connected commercial networks and enterprises. People started to use Internet for communication by electronic mail, instant messaging and other applications and services. Now, the Internet has significant impact on culture, commerce and technology.

In the last 5 years the volume of network traffic has grown 12 times [34] and a similar rate of growth is even expected to persist in next years. The significant increase of network traffic in recent years is mostly caused by new video on demand services and Internet of Things (IoT). It is predicted that more than 50 billions IoT devices will be connected to the Internet in 2020, which will produce large volumes of data and significantly increase the amount of traffic on network links to data centres.

As the amount of network traffic is growing very fast, large Internet service providers (ISPs) started to upgrade backbone links to 40 Gb and 100 Gb Ethernet. Data centres need to connect top of rack switches by 100 Gb links and call for 1 Tb Ethernet, because most of network links are fully saturated. The increasing speed of network links has a direct impact to the architecture of network devices. Network devices must be faster and more powerful.

High-speed packet processing is important especially in network security and monitoring systems, where any packet drop can decrease precision of monitoring or avoid detection or mitigation of malicious traffic. Security systems have many time-critical operations. Fast pattern matching is used in Intrusion Detection Systems (IDS) [51, 19] to detect security threats, deep packet inspection is needed to analyse application protocols [28] and fast packet filters are needed to mitigate Distributed Denial-of-service attacks (DDoS) in scrubbing centres. All these operations are computationally intensive and current general purpose processors (CPUs) cannot guarantee for these operations wire-speed 100 Gbps throughput [12].

Fast packet processing is also required in modern data centres, where high power consumption has a direct impact to the operational cost. Therefore, the goal of every data centre is to decrease power consumption. FPGA based network interface cards can accelerate network applications and increase performance of single server. Then target application would require for the same performance less servers and power consumption is reduced. Therefore, data centres started to use FPGA based acceleration cards in order to scale up the performance and reduce power consumption [20, 50, 53]. These cards need high performance hardware architectures for wire-speed packet processing.

In recent years, the hardware acceleration in data centres is highly focused on Open vSwitch (OVS) [52], which is necessary to run network functions in the virtual environment, but has very low performance [31]. Packet capture and distribution to virtual machines consume a lot of processor cycles and slow down functions in virtual machines. The performance of OVS can be increased by hardware accelerated network interface cards [48] with autonomous distribution of packets to virtual machines. Unfortunately, such cards have to provide wire-speed packet parsing and header fields extraction (IP addresses, TCP/UDP ports, Protocols, etc.) together with the classification and distribution of packets to target virtual machines. As all these operations have to be implemented in the card, corresponding hardware architectures and cards have to be designed.

As application services in data centres have to respond to user requests in a very short time, network infrastructure has to be optimised not only to high throughput, but also to low latency. Current CPUs cannot achieve low latency and 100 Gbps throughput [12, 31]. To achieve wire-speed 100 Gbps throughput every packet has to be processed within only 5 ns. It means that a single CPU core has only a few instructions to process a packet, which is a strong limitation to build any system for 100 Gbps networks. Therefore CPUs cannot be used at these speeds for precise network monitoring, security systems, low latency devices or high performance OVS switches.

In order to achieve wire-speed 100 Gbps throughput, network systems have to utilize FPGA or ASIC technology. FPGA technology provides high performance and is highly configurable as well. ASIC chips are powerful, but they have limited flexibility. Once the ASIC chip is produced, the function cannot be modified. Nevertheless, the flexibility is essential for any network system, because network traffic processing is changing with every new protocol, application or service. Therefore, 40 Gbps network interface cards with FPGAs started to by deployed to data centers as a hardware platform for the acceleration [20] and will be probably more and more frequntly used in the future.

We can see that the demand for hardware acceleration is rising with the 100 Gbps Ethernet technology. New hardware architectures have to be designed to accelerate network functions for precise monitoring, security systems, OVS switches, services in data centres and other network applications. Moreover, these applications need new 100 Gbps FPGA based network interface cards to provide new hardware platform for the acceleration.

1.1 Research Area

The research is focused on high-speed network traffic processing, where hardware acceleration is needed to achieve wire-speed 100 Gbps throughput. In computer networks, there are many time-critical operations, which have to be performed on every input packet. The most frequently used time-critical operations are *packet parsing*, *packet classification*, *pattern matching* and also *deep packet inspection* (L7 analysis). Many hardware architectures [25, 59, 32, 60, 18, 62, 17, 29] have been introduced for packet processing in 10 Gbps networks. Unfortunately, these architectures are not able to scale up the processing speed to 100 Gbps and naive parallel processing causes packet reordering and large buffers. Thus new hardware architectures have to be designed. It is important to note that every packet has to be processed within only 5 ns to achieve wire-speed 100 Gbps throughput. Therefore, hardware architectures have to be highly optimised to processing speed.

Packet parsing is needed in every network device to analyse packet headers and extract important header fields (IP addresses, TCP/UDP ports, etc.). This time-critical operation has been accelerated on an FPGA by Brebner [5], but introduced architecture has a very

long latency and utilizes a lot of hardware resources. As packet parsing is only the first stage of any network traffic processing, it has to utilize only small portion of the FPGA chip. Therefore, it is necessary to reduce FPGA logic utilization and the amount of pipeline stages for latency sensitive applications.

Packet classification [33, 32, 60, 43, 6, 56] is widely used to control packet filtering, cropping and modification, but it can be used as well to measure statistics, redirect network traffic or for any other application working on the third network layer (L3). The operation is controlled by classification rules. To support many rules it is necessary to use an off-chip memory. Therefore, it is important to design a hardware architecture with wire-speed 100 Gbps throughput and utilize off-chip memory to support large ruleset.

Many network security and monitoring devices need to provide analysis of network traffic at the application level (L7). Pattern matching is often used to detect security threats, user identifiers or application protocols. Many hardware architectures [59, 25, 62, 70, 42, 11, 69] have been developed to accelerate pattern matching in IDS systems. As IDS systems have a large set of regular expressions, many papers [7, 11, 42, 62, 70] introduced reductions of memory or hardware resources in order to use a smaller FPGA or support more regular expressions. Multi-striding [18, 9] and spatial-stacking [67] techniques have been introduced to scale up the processing speed. Unfortunately, wire-speed 100 Gbps throughput hasn't been achieved.

As regular expressions have a limited descriptive power, more precise L7 analysis use libraries or modules implemented in C language [28]. These libraries are usually based on a more computational powerful model and cannot be easily mapped into the FPGA. Nevertheless, L7 analysis is time consuming and wire-speed 100 Gbps throughput can be achieved only with a hardware acceleration.

We can see that many network applications need hardware acceleration for 100 Gbps wire-speed processing. To increase the processing speed, new hardware architectures have to be designed for packet parsing, packet classification, pattern matching and deep packet inspection. These time-critical operations can accelerate for example Flow monitoring, IDS/IPS systems, systems with deep packet inspection, firewalls, load balancers, DDoS mitigation systems or OVS switches.

1.2 Research Objectives

The main research objective for this thesis is to

"design hardware architectures for wire-speed 100 Gbps packet processing in order to accelerate network applications and systems for large ISPs and data centres."

The hardware architectures must cover the most frequently used time-critical operations in computer networks and must be optimised for FPGA technology to process all network traffic on 100 Gbps link without any packet drop. Moreover, all designed architectures must provide high flexibility. High flexibility is needed especially for the processing of application protocols and also in security systems, where a fast response to new security treats is needed. Therefore, all hardware architectures must be easily customizable and configurable.

The second very important requirement is the low utilization of hardware resources. In order to accelerate network applications many network functions, have to be placed inside the FPGA at the same time. Therefore, the hardware architectures must be optimised to use as low FPGA resources as possible.

A very important requirement is to provide an FPGA based hardware platform, which ensures the feasibility of hardware acceleration using designed architectures. The goal is to provide technology for 100 Gbps networks to accelerate individual network functions and also complete systems for large ISPs or data centres.

1.3 Thesis Outline

The thesis is composed as a collection of papers. The research contribution of this thesis is thus constituted by six peer-reviewed research papers, in their original publication format. The thesis is organised as follows: Chapter 1 (this chapter) gives an introduction to the thesis. Chapter 2 surveys the state of the art, and presents relevant background information for the research. Chapter 3 summarises the research process and gives an overview over the papers constituting the research contribution. Finally, Chapter 4 presents conclusions and proposes future research directions.

Chapter 2

State of the Art

As the speed of network links is growing very fast, the infrastructure, security and monitoring tools need more and more computational resources. New algorithms and hardware architectures have to be designed to achieve wire-speed 100 Gbps packet processing. We can see in Figure 2.1 that the time to process packets is decreasing with the speed of network links. To achieve 100 Gbps speed, network applications have to process every packet in less than 5 ns, which corresponds to 18 CPU cycles for CPU running at 3.6 GHz.

Current processors are not able to process all network traffic at wire-speed on 10, 40 or 100 Gbps links. In order to scale up the processing speed, network devices and applications often use a hardware acceleration. The hardware acceleration is usually focused on time-critical operations, which consume most of the processor time and decrease a throughput of network applications and systems. Therefore our research is focused on a hardware acceleration of five most widely used time-critical operations in computer networks:

Packet Parsing and Header Field Extraction – has to be performed by all network devices in order to gather a specific information from network packets like IP addresses, Ports, beginning of L7 layer etc. As the speed of network links is increasing rapidly, a high speed packet parsing is required. Many network devices perform the packet parsing by multi-core network processors. Unfortunately, 100 Gbps throughput can be achieved only if the packet parsing is done in a hardware [17, 37, 27, 39, 5, 54].

Longest Prefix Matching (LPM) – refers to algorithms used by many network devices to select the most specific table entry (prefix), which matches an IP address of the input packet. The LPM operation is typically being performed by routers to match

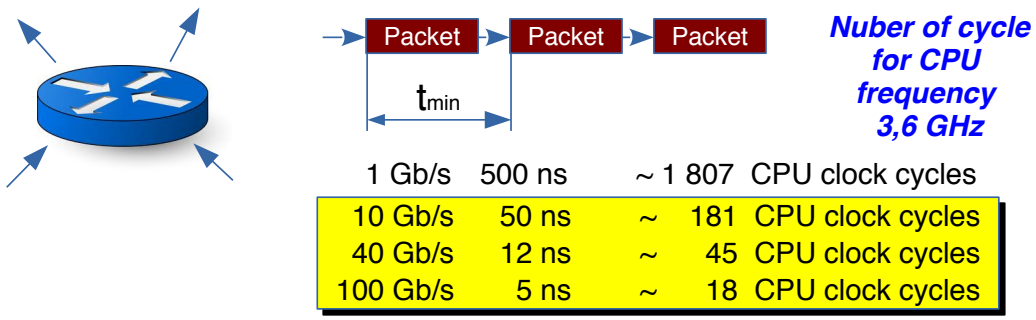

Figure 2.1: Time to process the shortest packet for different speeds of network links.

the most specific entry in a forwarding table, but it can be utilized by firewalls and other devices. Many hardware architectures have been designed to accelerate the longest prefix matching [30, 61, 44, 58, 47], but the IPv6 protocol and a growing speed of network links require new high-speed LPM algorithms with low memory utilization.

Packet Classification – is used in many network devices such a firewalls, IDS or IPS systems. During the classification packets are matched with a set of rules, which are usually defined by values, ranges or prefixes of packet header fields. Generally, the classification is a mathematical problem of a multidimensional range search. Due to the rule set size and complexity of rules, it is very difficult to match all rules in less than 5 ns and achieve 100 Gbps throughput. Many hardware architectures have been designed to accelerate the packet classification [64, 32, 60, 45, 29, 55, 40].

Pattern Matching – matches a set of patterns in a packet payload. The patterns are often described by strings or by regular expressions. It is a time-critical operation widely used for an identification of application protocols or a detection of malicious traffic in IDS/IPS systems [19, 51]. Current processors are not able to achieve gigabit speed [12] for a large set of strings or regular expressions, even if they use the fastest algorithms. Therefore, many hardware architectures [59, 25, 62, 70, 42, 11, 69] have been proposed to accelerate the string or regular expression matching.

Deep Packet Inspection – is required for a precise analysis of network traffic at the application level. Although the pattern matching has been widely adopted by security systems, the descriptive power of strings or regular expressions is limited. Therefore, Deep Packet Inspection (DPI) libraries [28] started to use software modules implemented in a C language. These modules need for the high-speed packet processing a hardware acceleration, but cannot be easily implemented in a hardware. As the software implementation of modules can be easily adjusted to the target application and a hardware can increase processing speed, a new hardware/software codesign concept would be appropriate for a precise DPI acceleration.

Time-critical operations have been addressed by many research papers. The common goal is to increase the processing speed with reasonable hardware resources. The following sections describe the state-of-the-art for all five operations and provide an information about the scalability of hardware architectures to 100 Gbps throughput.

2.1 Packet Parsing

The packet parsing performs an analysis of packet headers in order to extract header fields for a further packet processing (IP lookup, packet filtering, etc.). The processing is shown in Figure 2.2. The input is a packet and the output is an extracted set of header fields. The analysis is driven by the description of protocols, which is usually transformed to the configuration of packet parser. The configuration is stored in the memory or hardwired in the architecture.

The first packet parser has been introduced by Braun et al. [17], who employs the onion-like structure of hand-written protocol wrappers to parse packets. However, due to the 32-bits-wide data path and an old FPGA, the parser achieves only 2.6 Gbps throughput.

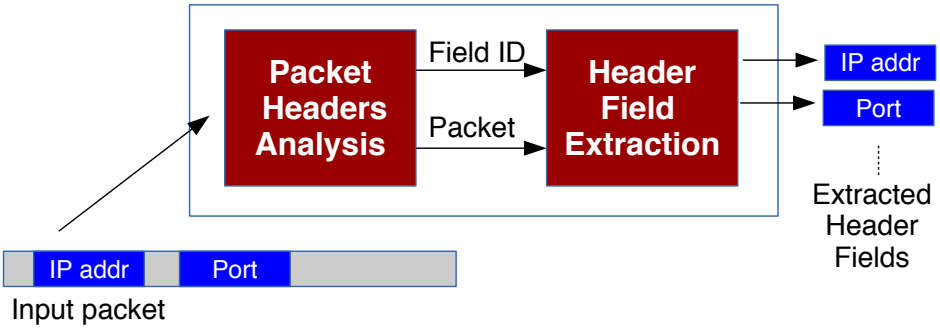

Figure 2.2: Packet parsing and header field extraction.

There is no extensive concept of a common interface for module reuse. It is unclear how the parser scales for a wider data path in order to achieve a higher throughput.

Dedek et al. [27] utilizes a high-level Handel-C language to describe the process of packet parsing, but implementation details are not disclosed. The reported speed of 1 454 Mbps implies that a rather narrow data bus (probably 16 bit) was used. Therefore, the concern is about the scalability in terms of both an effective description in a Handel-C language and an effective compilation to a hardware for much wider data words. This work also demonstrates that using processors for the packet parsing gives poor results. Compared to the Handel-C implementation, a custom RISC processor designed specifically for the packet parsing occupies roughly the same chip area, but achieves only a half of the throughput. A solution based on the MicroBlaze [3] processor (which is not optimized for the packet parsing) requires double resources and brings only 5.7 % throughput compared to the Handel-C solution.

Attig and Brebner [5] have utilized domain-specific Packet Parsing (PP) language to describe the structure of packet headers and the methods specifying the parsing rules. The description is then compiled from the PP language to a highly pipelined implementation. However, the results indicate that the price for a convenient design entry is the chip area and the latency. Most parsers with 1024-bit datapath use over 10 % of the resource-abundant Xilinx Virtex-7 870HT FPGA [4] and the latency varies from 292 to 540 ns.

The Kangaroo system [39] uses RAM to store the packets and employs the on-chip CAM to perform a look ahead, which is a process of loading several fields from the packet memory at once, allowing to parse several packet headers in a single cycle. The dynamic programming algorithm is used to precompute data structures, so that the parsing of the longest paths in a parse tree is the most accelerated operation by the look ahead, as it is impractical to perform the look ahead for all the possible protocol combinations. This approach has the architectural limitation of storing the packets in the memory and accessing them afterwards. The memory soon becomes a bottleneck.

Although many hardware architectures have been presented to accelerate the packet parsing, only Brebner introduced a hardware architecture for 100 Gbps links. Unfortunately, the architecture has a very high latency and utilizes a lot of hardware resources. Therefore, it is necessary to focus further research on the design of new architectures that will be highly optimized for the utilization of hardware resources, while providing a sufficient flexibility and a low latency.

2.2 Longest Prefix Matching

Longest prefix matching (LPM) is a time-critical operation, which is used in routers to determine an output port based on the packet destination IP address. This operation looks up an entry in a forwarding table and provides as a result the entry containing the longest prefix equal to the packet destination IP address. LPM is used not only in routers, but also in packet filters, load balancers, OpenFlow switches and next generation firewalls.

Routers and other networking devices have to process 156 millions packets per second to achieve wire-speed 100 Gbps throughput. It means that the LPM results have to be produced every 5 ns, which is impossible to achieve without a dedicated hardware [58]. However, such architectures usually suffer from slow and energy intensive accesses to an external memory. In order to achieve wire-speed 100 Gbps throughput and a low power consumption, it is necessary to store the forwarding table in an on-chip memory.

The majority of LPM algorithms is based on a *trie* data structure. It encodes a set of prefixes from a forwarding table into a binary tree. Each node of the tree has up to two pointers to child nodes where left and right child nodes represent prefixes created from the parent's prefix by appending 0 and 1, respectively. LPM is then performed by traversing the trie from the root to leaves according to bit values of a packet's destination address taken from the most significant bit to the least significant bit. The last prefix node visited during such traversal represents the longest matching prefix.

The trie data structure is well designed to implement adding, removing and matching operations. However, because of the high number of pointers, the trie is not a memory efficient representation of a prefix set. Moreover, it does not scale well with the length of the destination IP address, because it allows processing of only one input bit in each step. An example of the trie data structure is shown in Figure 2.3. It can be seen that the depth of the tree corresponds to the longest prefix in the data set. The maximum depth is 32 nodes for the IPv4 and 128 nodes for the IP6. If the algorithm checks at once only one node, many processing steps are needed to match one IP address and 100 Gbps throughput cannot be achieved. To increase processing speed multiple trie nodes (bits) have to be checked at once in a single step.

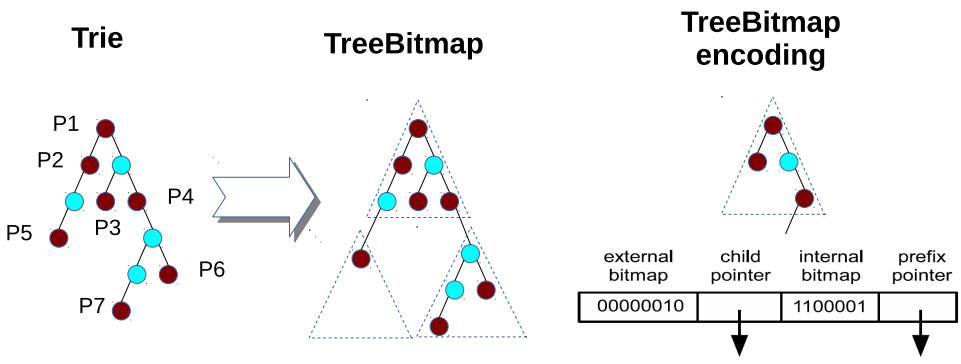

Figure 2.3: Tree Bitmap mapping and encoding.

In order to allow processing of multiple input bits per a step, multibit tries have been proposed. *Tree Bitmap* (TBM) [30] is one of the best known implementations of the multibit trie approach. Prefixes from a forwarding table are within TBM stored in a 2^{SL} tree, where each TBM node can contain up to $2^{SL}-1$ trie nodes. The parameter SL is called the stride

length and specifies the number of input bits processed in each step. Mapping TBM nodes with $SL = 3$ to the trie is shown in Figure 2.3 together with the TBM node structure.

The node contains two pairs consisting of a bitmap and a pointer, which allow to access ordinary child (external) or prefix (internal) related information. Such a compact representation allows the node to be read from a memory in a just one clock cycle. Moreover, bitmaps make TBM easy to implement in a hardware. The fixed structure of the node also simplifies performing incremental updates of a forwarding table. However, it may introduce high memory overhead, especially in a sparse prefix tree.

Shape Shifting Trie (SST) [61] is another multibit trie algorithm. It is based on TBM, but reduces memory overhead introduced by TBM, when representing a sparse prefix tree. Instead of having nodes with a fixed structure, SST allows nodes to adapt to a structure of an underlying trie. This adaptability is allowed by another bitmap (shape bitmap) introduced in a node's representation (see Figure 2.4) and is constrained only by the parameter K, which specifies the maximum number of trie nodes represented by the SST node. Even though SST shows very low memory demands, its computational complexity is usually unacceptable. Moreover, to the best of our knowledge, there is no hardware architecture for SST.

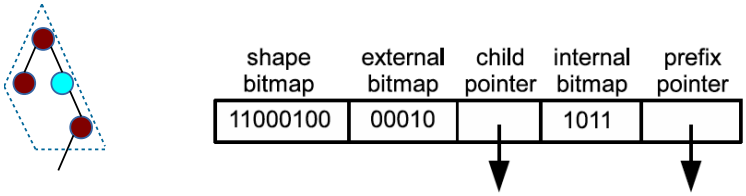

Figure 2.4: Shape Shifting Trie node encoding.

The LPM architecture for 100 Gbps networks with currently the lowest memory demands has been described in [44]. This algorithm, which will be further referred to as Prefix Partitioning Lookup Algorithm (or PPLA), also uses the trie data structure. However, the trie is utilized only for partitioning a set of prefixes into several disjoint subsets, which are stored in separate binary search trees or 2–3 trees, each of them processed in a separate processing pipeline. The PPLA has good memory efficiency (1 B of memory for storing 1 B of IPv4 or IPv6 prefix) [44], but building this internal representation is connected with a very high pre-processing overhead. Moreover, memory demands of PPLA grow linearly with the number of stored prefixes, which is more than memory demands of trie-based LPM algorithms. When prefixes are stored in a data structure based on the trie or in the trie itself, nodes close to the root are shared by several prefixes. Then less memory resources are utilized to store a forwarding table.

To summarize, recent LPM architectures are able to achieve 100 Gbps throughput, but only at the cost of a very high pre-processing overhead and without any compression or memory optimisation provided by the trie data structure. An effective compression of a prefix set is important to store more prefixes and support large forwarding tables. Therefore, it is necessary to analyse properties of trie, TBM, and SST data structures and design a memory-optimised LPM hardware architecture with 100 Gbps throughput.

2.3 Packet Classification

The aim of the packet classification is to split packets into different classes based on the set of rules. The input of the algorithm is a set of classification rules ordered by priority and the values extracted from packet header fields. The operation has to find the rule, which matches extracted header fields. If the packet matches multiple rules, the result is selected by priority. The process of packet classification is shown in Figure 2.5. The extracted packet header fields are green and the classification rules are red.

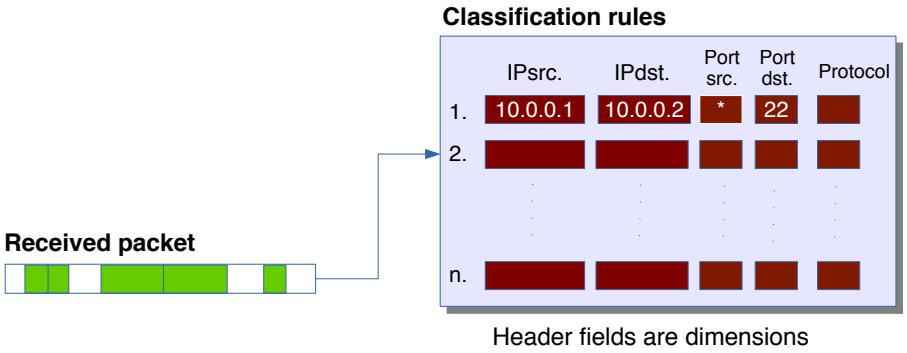

Figure 2.5: Packet classification in multiple dimensions.

Ternary associative memory (TCAM) is a high performance and easy to use hardware device, which is often used to speed up the packet classification. Due to the massive parallel comparisons at the hardware level, the TCAM memory can provide results with a constant time complexity and is able to achieve high throughput. On the other hand, TCAMs have very limited capacity, high power consumption and high cost per bit compared to conventional SRAM memories. Therefore, many hardware architectures utilize SRAM memories together with a hardware implementation of appropriate classification algorithm.

The packet classification can be viewed as a geometric problem, where each packet header field is one dimension of a discrete multidimensional space. The classification rules are represented in the space as rectangular objects and individual packets are represented as points. Then the goal of the packet classification is to find in the search space geometric objects, which cover the input packet.

The geometric representation of packet classification is utilized by HiCuts [32] and HyperCuts [60] algorithms. These algorithms use decision trees to find the classification rule. Every inner node of the decision tree divides the search space using planar surfaces and the leaf node contains the matched rule. The time complexity and memory requirements depend on the internal structure of decision tree, especially on the number of divisions in every inner node. HyperSplit algorithm [57, 56] employs the decision tree to split the search space in inner nodes only into two parts. This optimisation reduces the number of inner nodes at the cost of higher decision tree depth. The goal of these algorithms is to reduce memory requirements in order to store the whole decision tree into the on-chip memory and use pipelined processing with multiple memories to achieve 100 Gbps throughput. Unfortunately, network applications need many classification rules, which can be stored only in large external memories. HiCuts, HyperCuts and HyperSplit algorithms are designed to reduce memory requirements, but they need to process a lot of inner nodes to classify a packet. So, if the decision tree is stored in an external memory, multiple

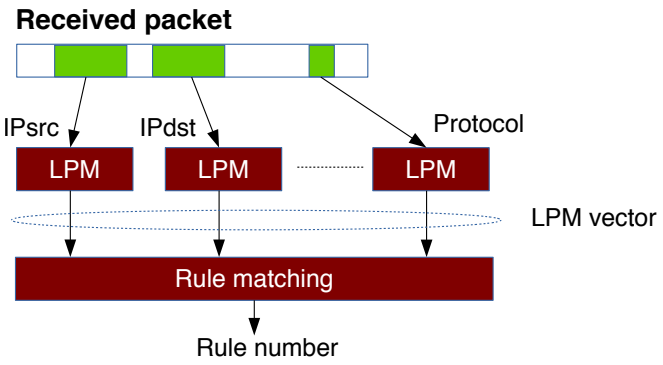

Figure 2.6: Basic scheme of decomposition algorithms.

memory accesses are needed and the processing is slow down significantly. It means that 100 Gbps throughput cannot be reached, if these algorithms use an off-chip memory.

Large off-chip memories can be well utilized by decomposition algorithms. These algorithms require only a few memory accesses and can achieve constant time complexity. Therefore, the number of classification rules can be easily increased by an off-chip memory without any significant impact to the processing speed. In decomposition methods, the packet classification is divided into several steps (or pipeline stages). Figure 2.6 shows the basic scheme of decomposition algorithms.

We suppose that the input of packet classification is a vector of packet header fields. LPM operation is a first step, which is performed for every packet header field independently. Each LPM search engine returns one item from the *prefix set*, which is a set of all possible LPM results for the given dimension. The result of LPM Stage is an *LPM vector*, containing one prefix for each dimension. After the LPM, all fields of the resulting LPM vector must be processed in some way (this is specific for each algorithm) to find the correct rule number. Key issue is that the number of possible LPM vectors can be extremely large. This is because all possible values of LPM vector are obtained by creating the Cartesian product of prefix sets.

The basic Cartesian product algorithm [63] precomputes a Cartesian product table, which contains resulting rule numbers for all possible LPM vectors. Because of the multiplicative nature of the Cartesian product, this table may become extremely large.

Other method of combining LPM results together is the Distributed Crossproducting of Field Labels [64]. LPM is modified to return all valid prefixes (not only the longest one) for a given field value. What follows is the hierarchical structure of small crossproduct engines. Inputs of each engine are two sets of prefixes (or Labels, in general). The engine then performs a set membership query for each possible pair of Labels. The result of engine is another set of Labels. The result of the last engine is in fact a set of rules, from which the one with the highest priority is selected.

Multi Subset Crossproduct Algorithm [29] brings further improvements to decomposition methods. Authors of this work replace Cartesian products by *crossproduct-rules*. These rules have significant impact to the memory requirements. Therefore, authors provide heuristics on how to break a ruleset into several subsets, eliminating the majority of crossproduct-rules. The paper also identifies rules that generate excessive amount of crossproduct-rules. These rules are called *spoilers* and are treated in a separate algorithm branch (in a hardware implementation, spoilers are moved to the small on-chip TCAM) to

further reduce the number of crossproduct-rules. The LPM operation is slightly modified to return a result for each subset, because subsets may contain different prefixes. A Bloom filter [15] is associated with each subset to perform the set membership query. If the Bloom filter output is true, one rule table memory access is performed to retrieve the resulting rule or crossproduct-rule.

We can see that all algorithms with decision trees require a lot of processing steps and cannot achieve 100 Gbps throughput without pipelined processing. As every pipeline stage has to read inner nodes from a memory, many memory accesses are needed to classify a packet and a large off-chip memory cannot be used without significant slow down of the processing. Decomposition methods can utilize an off-chip memory without performance limitations. So, the number of classification rules can be significantly increased. Unfortunately, MSCA algorithm has to store in an off-chip memory many crossproduct-rules, which consume the capacity of memory. Moreover, wire-speed throughput cannot be guaranteed. Therefore, new fast and memory optimised algorithms are needed to support large rulesets and 100 Gbps throughput.

2.4 Pattern Matching

Regular Expression (RE) matching is a widely used operation in computer networks for identification of application protocols, detection of network attacks, application aware load balancing and many other network applications. Current processors are not powerful enough to achieve 100 Gbps throughput [12]. The throughput of one processor core is limited to less than one Gbps. The matching speed can be increased to hundreds of Gbps only at the cost of a large number of processor cores. Although network processors have dedicated hardware units for RE matching, these units have usually significantly lower throughput [12, 49] than the capacity of network links. In order to achieve 100 Gbps throughput, it is more efficient to use a hardware acceleration using FPGA technology.

Many hardware architectures have been designed to accelerate the pattern matching for IDS systems[59, 25, 62, 70, 42, 11, 69], where thousands of RE have to be matched against network traffic. Several architectures take advantage of massive parallel processing in FPGA technology and use mapping of Non-deterministic Finite Automaton (NFA) to FPGA [59, 25, 26, 62, 70]. To achieve linear time complexity, all non-deterministic paths are processed in FPGA simultaneously. As the number of REs in IDS systems increases in time, many optimizations have been introduced to reduce FPGA logic utilization and map more REs to FPGA [62, 46]. Most of these optimizations are focused only on REs in IDS systems Snort [19] and Bro [51].

Prasanna has introduced a modular RE-NFA architecture[67, 68], which can be converted automatically into a modular circuit on FPGA. The circuit can be created from a set of REs without synthesis and can be uploaded to the FPGA by dynamic reconfiguration. The dynamic reconfiguration is also used for a fast update of the RE set in Dynamic BP-NFA [35].

The architectures based on a Deterministic Finite Automaton (DFA) [42, 11, 69] use a memory to store the transition table, which enables even faster update of the pattern set. Then the update doesn't require FPGA reconfiguration, because the hardware architecture remains the same. Only memory content is updated, if the RE set is changed. On the other hand, the construction of DFA from NFA has an exponential time complexity and can cause an exponential growth of states and transition table size, which has direct impact to memory requirements. Therefore, memory requirements have been reduced by a delayed

input DFA [42], a compression [7, 11] and other optimizations [69, 41]. Several architectures employ a combination of NFA and DFA[38, 8, 10] automata to cope with large data sets and an exponential growth of DFA states caused by $.*$ constructions in REs.

Most hardware architectures provide reductions of memory requirements or FPGA logic utilization to match more REs. With the growing speed of network links, it is necessary also to increase the matching speed. For the pattern matching, the speed depends on the frequency and the number of bytes processed per clock cycle. As the FPGA frequency increases only slightly over time, it is necessary to process more bytes at once.

Brodie presented the first architecture with accepting multiple bytes per clock cycle [18]. Prasanna introduced spatial stacking for multi-character matching [67] with the RE-NFA architecture, but high fan-out of final circuit significantly decreases the frequency, even for matching only 8 bytes per clock cycle. Unfortunately, more than 64 bytes have to be processed at once to achieve 100 Gbps throughput. Becchi introduced a multi-striding technique [9], which can be applied for NFA or DFA automata and is widely used to increase throughput of RE matching architectures.

Multi-striding can increase the processing speed of DFA based architectures only at the cost of higher memory requirements, which cannot be implemented in current chips. NFA based architectures can be speed up by multi-striding only at the cost of large amount of hardware resources. With the size of input symbols, the size of automaton grows nearly exponentially and the frequency drops down dramatically. Due to the frequency drop, we were not able to create a hardware architecture with more than 40 Gbps throughput. The same problem was with the spatial stacking technique.

We can see that neither multi-striding nor spatial stacking technique are able to scale the throughput of a single automaton to 100 Gbps. Therefore, it is necessary to design a new high-speed pattern matching architecture, which can scale up the throughput over 100 Gbps and can be implemented in current FPGAs.

2.5 Deep Packet Inspection

Network security and advanced monitoring applications with the application layer analysis usually perform a deep packet inspection (DPI) of interesting packets in the packet payload. Many papers deal with hardware acceleration of IDS system Snort [19], which has restricted focus to the regular expression matching only. However, the area of network security is much richer. If the DPI acceleration should be focused only on the regular expression matching, security systems would be strongly limited and insufficient for robust practical deployment.

L7-filter [1] relies similarly to Snort entirely on regular expressions. It is a Linux based packet classification software for identification of protocols at the application layer (L7). An open-source library for application layer traffic processing called an nDPI [28] provides an excellent example showing that the regular expression matching alone is not sufficient. While the nDPI library is probably too complex to be fully hardware accelerated, software modules can certainly offload a part of the work to the hardware in the sense of hardware/software codesign.

Shunting [65] is a hardware/software architecture that provides a lightweight mechanism for IPS systems to take advantage of the "heavy-tailed" nature of network traffic to offload work from the software to the hardware. The hardware can forward, drop or diverse network traffic according to the rules in hardware caches. The rules use IP addresses and connection 5-tuples (src/dst IP, src/dst Port, Protocol). Shunting has been designed for Bro [51] intrusion detection system. However, it accelerates only packet forwarding and filtering

without further possibilities for a hardware acceleration. Therefore, it is not very useful for the majority of network security and monitoring applications, because they need statistics and other information about every received packet.

SDNet environment [2] has been recently announced by Xilinx as a system for hardware accelerated networking defined by software. The system relies upon high-level language to describe a network application, which is then compiled to a form of a hardware accelerator for a Xilinx FPGA. From a limited information available at the time of writing, it seems that SDNet has limited throughput and it is not focused on a DPI acceleration. SDNet is a flexible tool that can be used to describe custom hardware modules a little bit easier. Network traffic processing can also be described in the P4 high-level language [16] and then mapped to the hardware accelerator. The P4 language has been designed as a high-level description of network traffic processing and forwarding. It enables protocol, vendor and target independent definitions. Thus, it can be used for more comfortable programming of hardware accelerators, but similarly to SDNet, P4 does not provide any concept or hardware architecture for a DPI acceleration.

It can be seen that none of the known state of the art approaches provides a software controlled hardware acceleration of DPI. Therefore, there is a valid need for the design of a novel acceleration concept for application layer analysis with 100 Gbps throughput, high flexibility and precise monitoring.

Chapter 3

Research Summary

The goal of this thesis is to propose new hardware architectures, which are able to process network traffic at 100 Gbps speed. Proposed hardware architectures address five most widely used time-critical operations in network applications and systems. The architectures provide state-of-the art technology for 100 Gbps networks and can be used as key building blocks for hardware accelerated network devices and systems.

Packet parsing and header field extraction are needed in every network device to get information from packet headers (src. and dst. IP addresses, src. and dst. port, etc.). Therefore the first research paper is focused on flexible packet header parsing. The parser is generated from XML description of network protocols. The XML is used also to specify header fields, which have to be extracted from packet headers. The architecture is very flexible, because it allows the users to change the set of extracted fields at runtime. Moreover, the result is provided as a compact data structure called the unified header, which can be easily sent over the network or PCIe interface for further processing. Unfortunately, one packet parser was able to process network traffic only at 20 Gbps. The 100 Gbps throughput was achieved only by multiple parallel paths (parsers), which need a lot of logic resources.

Therefore the second paper proposes a highly pipelined modular architecture. The architecture has very low latency and throughput over 100 Gbps. Due to the designed processing pipeline, the architecture utilizes much less hardware resources in comparison to previous approaches. Moreover, user can easily optimize the architecture to decrease latency or increase the throughput. Both parameters can be balanced to meet requirements of target application. The modular architecture allows to add new protocols very easily.

The paper III is focused on Longest Prefix Matching (LPM), which is used in every IP router and firewall. LPM is usually performed on destination IP address to find the longest prefix in forwarding table or in the set of filtering rules. The size of forwarding tables is steadily increasing. Current routers have tables with more than 500 thousands of IP prefixes and have to perform LPM for every packet. To achieve 100 Gbps speed, routers have to store forwarding tables in very fast and large memories. In paper III, we propose to use efficient compression to store forwarding table in the on-chip memory. The trie data structure is compressed to newly designed instructions. Fast processing is achieved by deep pipeline, where every pipeline stage executes one instruction to move from trie root node to the longest prefix. Due to the deep pipeline, the architecture has 100 Gbps throughput for IPv4 as well as for IPv6 addresses. The LPM is performed on destination IP address, which can be provided by any packet header parser described in paper I or II.

Packet classification is used in firewalls to match packets against filtering rules. In paper IV, we propose a high performance packet classification architecture, which employes

perfect hash functions to match filtering rule with constant time complexity. LPM is used as a first step to support rules with IP prefixes and ranges. External SRAM memories are utilized to store large perfect hash table and support large ruleset. The algorithm performs a decomposition to LPM and rule matching to cope with IP prefixes, but the decomposition can cause crossproduct rules [29, 55], because LPM provides only the longest prefix and we can have rule with shorter one. Therefore, crossprodact rules have to be considered and added to the ruleset. Previous architectures use bloom filters to reduce the amount of crossproduct rules, but still memory requirements remain high. Therefore we utilize the perfect hash table to map crossproduct rules to the correct rule number. It means that the crossproduct rule is stored only in the perfect hash table and the rule table contains only original rules. This way a significant amount of memory is reduced. The architecture is pipelined and it requires only two QDR SRAM to achieve 100 Gbps speed. Moreover, the throughput can be further increased by more off-chip memories.

The longest prefix matching and packet classification operate over packet header fields extracted by packet header parsers, which works up to network layer (L3). But network security and monitoring applications also need a deep packet analysis at the application layer (L7). IDS systems use for L7 analysis pattern matching. Therefore, many research papers are focused on hardware acceleration of pattern matching for IDS systems. These architectures are optimized for large set of regular expressions, but they are not able to scale the processing speed to 100 Gbps. The throughput of hardware architectures has been increased by multi-striding [9, 18] and spatial-stacking [67], but both techniques are able to scale the throughput only to tens of Gbps. Therefore, we propose in the paper V pipelined automata architecture, which can scale pattern matching throughput over 100 Gbps.

The architecture consists of multiple pipelined automata directly connected to one shared input buffer. Automata states circulate in the pipeline to input data words, which is much more effective than multiplexing of the input data to parallel automata. Processing speed can be easily increased by the number of automata in the pipeline at the cost of only linear growth of FPGA and memory resources. As automata only use local connection to the input buffer and neighbouring pipeline stages, the frequency of the architecture is not decreased with the number of automata. Therefore, the throughput can scale to hundreds of Gbps. Moreover, the architecture uses for all packets only one shared input buffer, which significantly reduces memory requirements of the architecture.

Unfortunately, regular expressions have only limited descriptive power. Precise L7 analysis based on a more powerful computational model needs libraries or modules implemented in C languge [28], which cannot be easily mapped into the FPGA. Every application protocol has different syntax and for every application, we need different type of hardware acceleration. It is well known that hardware development takes much more time than software implementation and it is not possible to create a special hardware architecture for every protocol or application. Therefore, we have designed in paper VI Software Defined Monitoring (SDM) concept, which takes advantage of hardware performance and software flexibility for L7 analysis. The software is able to control precisely the level of processing in the hardware. For every flow, we can select a different type of processing. As most of the information are in first packets of the flow, software can perform deep analysis in first N packets of the flow and remaining packets can be offloaded to hardware only to count statistics, provide packet header fields or simply drop the packet. Moreover, software can select the level of information according to the processor load. We have shown that the SDM concept is able to decrease the CPU load to less than 20 %.

The SDM concept uses L7 parsers on CPU cores and is able to provide precise Netflow/IPFIX data together with the information from application protocols. To reduce communication among CPU cores, it is important to recognize application protocols in hardware and send packets to the right L7 parser (CPU core). Therefore, the SDM concept has been extended by pattern matching architecture in the paper V for identification of application protocols (HTTP, SIP, etc.) and distribution of packets to corresponding CPU cores.

All papers provide hardware architectures for 100Gbps networks, which can be used as a key build block of any network security and monitoring device or application. The acceleration considers analysis and filtration of network traffic up to layer 7 of ISO/OSI model. The hardware architectures has been used in network monitoring probes in CESNET academic network and create technology, which has been transferred to commercially successful products.

3.1 Papers

This chapter describes the most relevant details for all papers included in this thesis. For each paper, we present the motivation, contribution and original abstract.

3.1.1 Paper I

Packet Header Analysis and Field Extraction for Multigigabit Networks

The aim of the paper is to provide highly flexible architecture for packet parsing and header field extraction with processing speed over 10 Gbps. The aim was also to reduce FPGA logic utilization and balance hardware resources and processing speed. Therefore the HDL code generator has been designed to achieve high flexibility together with high performance. The generator creates from the XML description of protocols a corresponding automaton, which is then transformed to optimized hardware architecture. To increase the processing speed the automaton has been improved to accept multiple symbols within one clock cycle. It means that the designer can balance the hardware resources and processing speed. Moreover, the XML description of protocols allows to add or change supported protocols very easily.

The result of the work is a highly flexible packet header parser. As the HDL code of the parser is generated from the XML description, the architecture is optimized to provide high processing speed and support protocols specified in the XML. The paper provides 20Gbps throughput for Virtex5 FPGA. For current Virtex-7 FPGAs, the processing speed can be increased to 100 Gbps by multiple parallel paths described in the paper.

Abstract

Packet header analysis and extraction of header fields needs to be performed in all network devices. As network speed is increasing quickly, high speed packet header processing is required. We propose a new architecture of packet header analysis and fields extraction intended for high-speed FPGA-based network applications. The architecture is able to process 20 Gbps network links with less than 12 percent of available resources of Virtex 5 110 FPGA. Moreover, the presented solution can balance between network throughput and consumed hardware resources to fit application needs. The architecture for packet header processing is generated from standard XML protocol scheme and is strongly optimised for

resource consumption and speed by an automatic HDL code generator. Our solution also enables to change the set of extracted header fields on-line without FPGA reconfiguration.

3.1.2 Paper II

Design methodology of configurable high performance packet parser for FPGA

The architecture presented in paper I is highly flexible and extensible, but for 100 Gbps throughput, it is necessary to use multiple parallel paths. The problem is that the frequency is decreasing with input data width. Therefore the aim of the paper II is to provide a highly pipelined hardware architecture instead of a flexible automaton. Deep pipeline allows to achieve throughput over 100 Gbps with a small amount of hardware resources because of high frequency. The flexibility is addressed by newly designed interfaces of protocol parsers (Ethernet, IPv4, IPv6, etc.), which are connected together and create a deep pipeline. Due to the well designed interfaces new protocols can be implemented easily.

The result is a highly pipelined hardware architecture for packet parsing and header extraction with 100 Gbps throughput in a single pipeline. The architecture allows to finely tune the latency and throughput and provides interfaces to add a new protocol parser.

Abstract

Packet parsing is among basic operations that are performed at all points of a network infrastructure. Modern networks impose challenging requirements on the performance and configurability of packet parsing modules. However, high-speed parsers often use a significant amount of hardware resources. We propose a novel architecture of a pipelined packet parser for FPGA, which offers low latency in addition to high throughput (over 100 Gb/s). Moreover, the latency, throughput and chip area can be finely tuned to fit the needs of a particular application. The parser is hand-optimized thanks to a direct implementation in VHDL, yet the structure is uniform and easily extensible for new protocols.

3.1.3 Paper III

Memory Efficient IP Lookup in 100 Gbps Networks

The aim of the paper is to provide a hardware architecture for 100 Gbps LPM, which is an operation widely used in routers, firewalls and other networking devices. To achieve 100 Gbps throughput, LPM has to perform for every IP lookup a lot of memory accesses. Therefore it is necessary to store all IP prefixes in the fast on-chip memory. The paper analyses available IPv4 and IPv6 routing tables and introduces new highly efficient compression of trie data structure, which represents all IP prefixes. The paper introduces new instruction set to encode the most frequent shapes of the trie data structure and presents hardware architecture, where instructions are executed in the deep processing pipeline. The compression allows to store all available IPv4 and IPv6 prefixes in the on-chip memory. Due to the deep processing pipeline, the presented architecture can achieve high frequency and provides 100 Gbps throughput.

The result is the hardware architecture for LPM in 100 Gbps networks. It has wire-speed throughput and can be used in any router, firewall or network security device. The proposed compression significantly reduces memory requirements to store trie data structure (routing table, etc.)

Abstract

The increasing number of devices connected to the Internet together with video on demand have a direct impact to the speed of network links and performance of core routers. To achieve 100 Gbps throughput, core routers have to implement IP lookup in dedicated hardware and represent a forwarding table using a data structure, which fits into the on-chip memory. Current IP lookup algorithms have high memory demands when representing IPv6 prefix sets or introduce very high pre-processing overhead. Therefore, we performed analysis of IPv4 and IPv6 prefixes in forwarding tables and propose a novel memory representation of IP prefix sets, which has very low memory demands. The proposed representation has better memory utilization in comparison to the highly optimized Shape Shifting Trie (SST) algorithm and it is also suitable for IP lookup in 100 Gbps networks, which is shown on a new pipelined hardware architecture with 170 Gbps throughput.

3.1.4 Paper IV

Fast and scalable packet classification using perfect hash functions

Packet classification is a time-consuming operation, which is used in every firewall, IDS/IPS system, OVS switch or DDoS scrubbing centre. The aim of the paper is to provide a packet classification algorithm with constant time complexity to achieve 100 Gbps throughput. The paper introduced how to use perfect hashing for packet classification and support large ruleset using off-chip QDR SRAM memory. All packet classification algorithms have to cope with crossproduct rules, if prefixes or range checking is supported and LPM is used as the first step of the algorithm. To reduce memory requirements, the crossproduct rules are stored only in the perfect hash table, which provides a correct rule number in the original ruleset. Moreover, it is possible to increase processing speed and the number of rules by utilising more off-chip memories. Only one memory is needed to achieve 100 Gbps throughput.

The result of the paper is the packet classification algorithm with constant time complexity and 100 Gbps throughput with only one QDR SRAM memory. The throughput can be further increased by more memories. The architecture supports large rulesets, because crossproduct rules are stored only in the perfect hash table.

Abstract

Packet classification is an important operation for applications such as routers, firewalls or intrusion detection systems. Many algorithms and hardware architectures for packet classification have been created, but none of them can compete with the speed of TCAMs in the worst case. We propose new hardware-based algorithm for packet classification. The solution is based on problem decomposition and is aimed at the highest network speeds. A unique property of the algorithm is the constant time complexity in terms of external memory accesses. The algorithm performs exactly two external memory accesses to classify a packet. Using FPGA and one commodity SRAM chip, a throughput of 150 million packets per second can be achieved. This makes throughput of 100 Gbps for the shortest packets. Further performance scaling is possible with more or faster SRAM chips.

3.1.5 Paper V

High-speed Regular Expression Matching with Pipelined Automata

The pattern matching is widely used in network security systems to detect attacks or malicious traffic. Many hardware architectures have been designed to accelerate IDS systems to match thousands of regular expressions in packet payload. To scale up the throughput many architectures use multi-striding or spatial stacking techniques. Unfortunately, both techniques are not able to scale up the processing speed to 100 Gbps. The aim of the paper is to provide a hardware architecture, which is able to scale the pattern matching throughput to hundreds of Gbps. In order to increase the processing speed, the architecture employs multiple pipelined finite state machines (FSM). Instead of multiplexing the data from one input packet buffer to corresponding FSM, all FSMs are connected to one pipeline and current states of FSMs circulate in the pipeline. It means that every FSM processes only one input symbol of the packet and passes the current state to the next FSM to process the next input symbol. The architecture has only local interconnections. Every FSM is directly connected only to the input buffer and two neighbouring FSMs. Therefore, the maximal frequency is not affected by the number of FSMs and the architecture is well scalable. Throughput can easily be increased by a larger input buffer and more FSMs.

The result is the Pipelined Automata architecture, which can be used to scale the pattern matching throughput to hundreds of gigabits. The Pipelined Automata is a complementary technique to the multi-striding or spatial stacking. It is designed to achieve high frequency and utilize low hardware resources.

Abstract

Pattern matching is a complex task which is widely used in network security monitoring applications. With the growing speed of network links, pattern matching architectures have to be improved in order to retain wire-speed processing. Multi-striding is a well-known technique on how to increase throughput of pattern matching architectures. In the paper we provide an analysis of scalability of multi-striding and show that it does not scale well and cannot be used for 100 Gbps throughput because utilization of FPGA resources grows exponentially. Therefore, we have designed a new hardware architecture for high-speed pattern matching that combines the multi-striding technique and parallel processing using pipelined finite state machines (FSMs). The architecture shares a single packet buffer for all parallel FSMs. Efficient implementation of the packet buffer reduces the number of BlockRAMs to 18% when compared to simple parallel implementation. Instead of multiplexing input data, the architecture pipelines the states of FSMs. Such pipelined processing with only local communication has a direct positive impact on frequency and throughput and allows us to scale the architecture to hundreds of Gbps.

3.1.6 Paper VI

Software Defined Monitoring of Application Protocols

Current security threats and network performance issues are moving to the application layer (L7). The application protocol parsing is a time critical operation, which cannot be easily accelerated in hardware. The design of complete L7 hardware processing would be too time consuming for any network security monitoring application. Moreover, network protocols are evolving very fast and hardware development is too slow to respond to new

features of network protocols. Therefore the paper introduces Software Defined Monitoring (SDM) concept, which provides a flexible hardware acceleration of L7 processing. The concept uses a hardware flow cache in the acceleration card, which is under the software control. Software can assign for every flow in the cache a level of processing in the hardware. The hardware card can export all packets of a given flow for software analysis, it can count Netflow/IPFIX statistics or just drop all packets of the flow. The concept allows software to selectively drop the level of information in the hardware and balance between the software analysis and hardware processing according to the processor load and requirements of target application.

The result is a new Software Defined Monitoring concept, which can be used in network security and monitoring devices. It is a powerful concept to accelerate deep packet inspection with high flexibility. Packet analysis is completely defined in software, which can utilize hardware to offload most of the traffic and balance the level of analysis according to the processor load and available hardware resources.

Abstract

Current high-speed network monitoring systems focus more and more on the data from the application layers. Flow data is usually enriched by the information from HTTP, DNS and other protocols. The increasing speed of the network links, together with the time consuming application protocol parsing, require a new way of hardware acceleration. Therefore we propose a new concept of hardware acceleration for flexible flow-based application level monitoring which we call Software Defined Monitoring (SDM). The concept relies on smart monitoring tasks implemented in the software in conjunction with a configurable hardware accelerator. The hardware accelerator is an application specific processor tailored to stateful flow processing. The monitoring tasks reside in the software and can easily control the level of detail retained by the hardware for each flow. This way the measurement of bulk/uninteresting traffic is offloaded to the hardware while the advanced monitoring over the interesting traffic is performed in the software. The proposed concept allows one to create flexible monitoring systems capable of deep packet inspection at high throughput. Our pilot implementation in FPGA is able to perform a 100 Gbps flow traffic measurement augmented by a selected application-level protocol parsing.

With the ongoing shift of network services to the application layer also the monitoring systems focus more on the data from the application layer. The increasing speed of the network links, together with the increased complexity of application protocol processing, require a new way of hardware acceleration. We propose a new concept of hardware acceleration for flexible flow-based application level traffic monitoring which we call Software Defined Monitoring. Application layer processing is performed by monitoring tasks implemented in the software in conjunction with a configurable hardware accelerator. The accelerator is a high-speed application-specific processor tailored to stateful flow processing. The software monitoring tasks control the level of detail retained by the hardware for each flow in such a way that the usable information is always retained, while the remaining data is processed by simpler methods. Flexibility of the concept is provided by a plugin-based design of both hardware and software, which ensures adaptability in the evolving world of network monitoring. Our high-speed implementation using FPGA acceleration board in a commodity server is able to perform a 100 Gb/s flow traffic measurement augmented by a selected application-level protocol analysis.

3.2 List of Publications

Papers Included in Thesis

I Petr Kobierský, Jan Kořenek, and Libor Polčák. Packet header analysis and field extraction for multigigabit networks. In *Proceedings of the 2009 12th International Symposium on Design and Diagnostics of Electronic Circuits&Systems*, DDECS, pages 96–101, Washington, USA, 2009. IEEE Computer Society. [30%]

II V. Puš, L. Kekely, and J. Kořenek. Design methodology of configurable high performance packet parser for fpga. In *Design and Diagnostics of Electronic Circuits Systems, 17th International Symposium on*, pages 189–194, April 2014. [10%]

III J. Matoušek, M. Skačan, and J. Kořenek. Memory efficient ip lookup in 100 gbps networks. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, Sept 2013. [33%]

IV Viktor Puš and Jan Kořenek. Fast and scalable packet classification using perfect hash functions. In *FPGA '09: Proceedings of the 17th international ACM/SIGDA symposium on Field programmable gate arrays*, New York, NY, USA, 2009. ACM. [30%]

V D. Matoušek, J. Kořenek, and V. Puš. High-speed regular expression matching with pipelined automata. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 93–100, Dec 2016. [30%]

VI L. Kekely, V. Puš, and J. Kořenek. Software defined monitoring of application protocols. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 1725–1733, April 2014. [20%]

Other Relevant Papers

- Roland Dobai, Jan Kořenek, and Lukáš Sekanina. Evolutionary design of hash function pairs for network filters. *Applied Soft Computing*, 56:173 – 181, 2017.
- Jiří Matoušek, Gianni Antichi, Adam Lučanský, Andrew W. Moore, and Jan Kořenek. Classbench-ng: Recasting classbench after a decade of network evolution. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems*, ANCS '17, pages 204–216, Piscataway, NJ, USA, 2017. IEEE Press.
- Lukáš Kekely, Jan Kučera, Viktor Puš, Jan Kořenek, and Athanasios V. Vasilakos. Software defined monitoring of application protocols. *IEEE Trans. Comput.*, 65(2):615–626, February 2016.
- D. Grochol, L. Sekanina, M. Žádník, J. Kořenek, and V. Košař. Evolutionary circuit design for fast fpga-based classification of network application protocols. *Appl. Soft Comput.*, 38(C):933–941, January 2016.
- R. Dobai and J. Kořenek. Evolution of non-cryptographic hash function pairs for fpga-based network applications. In *Computational Intelligence, 2015 IEEE Symposium Series on*, pages 1214–1219, Dec 2015.

- David Grochol, Lukáš Sekanina, Martin Žádník, and Jan Kořenek. *A Fast FPGA-Based Classification of Application Protocols Optimized Using Cartesian GP*, pages 67–78. Springer International Publishing, Cham, 2015.

- V. Košař and J. Kořenek. Towards efficient field programmable pattern matching array. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 1–8, Aug 2015.

- L. Kekely, V. Puš, P. Benáček, and J. Kořenek. Trade-offs and progressive adoption of fpga acceleration in network traffic monitoring. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2014.

- L. Kekely, M. Žádník, J. Matoušek, and J. Kořenek. Fast lookup for dynamic packet filtering in fpga. In *Design and Diagnostics of Electronic Circuits Systems, 17th International Symposium on*, pages 219–222, April 2014.

- V. Košař and J. Kořenek. On nfa-split architecture optimizations. In *Design and Diagnostics of Electronic Circuits Systems, 17th International Symposium on*, pages 274–277, April 2014.

- J. Kaštil, V. Košař, and J. Kořenek. Hardware architecture for the fast pattern matching. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2013 IEEE 16th International Symposium on*, pages 120–123, April 2013.

- V. Košař, M. Žádník, and J. Kořenek. Nfa reduction for regular expressions matching using fpga. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 338–341, Dec 2013.

- J. Matoušek, M. Skačan, and J. Kořenek. Towards hardware architecture for memory efficient ipv4/ipv6 lookup in 100 gbps networks. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2013 IEEE 16th International Symposium on*, pages 108–111, April 2013.

- Viktor Puš, Lukáš Kekely, and Jan Kořenek. Low-latency modular packet header parser for fpga. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 77–78, New York, NY, USA, 2012. ACM.

- V. Puš and J. Kořenek. Reducing memory in high-speed packet classification. In *2012 8th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 437–442, Aug 2012.

- V. Košař and J. Kořenek. Reduction of fpga resources for regular expression matching by relation similarity. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 401–402, April 2011.

- V. Puš, M. Kajan, and J. Kořenek. Hardware architecture for packet classification with prefix coloring. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 231–236, April 2011.

- Viktor Puš, Jiří Tobola, Vlastimil Košař, Jan Kaštil, and Jan Kořenek. Netbench: Framework for evaluation of packet processing algorithms. In *Proceedings of the 2011*

ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, ANCS '11, pages 95–96, Washington, DC, USA, 2011. IEEE Computer Society.

- J. Tobola and J. Kořenek. Effective hash-based ipv6 longest prefix match. In *Design and Diagnostics of Electronic Circuits Systems (DDECS), 2011 IEEE 14th International Symposium on*, pages 325–328, April 2011.

- J. Kaštil and J. Kořenek. Hardware accelerated pattern matching based on deterministic finite automata with perfect hashing. In *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*, pages 149–152, April 2010.

Chapter 4

Discussion and Conclusions

This chapter summarises all research results and presents the deployment and commercialization of new hardware architectures. It provides a discussion of obtained results and conclusions together with new directions for future work.

Thesis presented new hardware architectures capable of processing the network traffic at 100 Gbps speed using FPGA technology. For the hardware acceleration, we have selected the most widely used time critical operations: (i) *packet parsing and header field extraction*, (ii) *longest prefix matching*, (iii) *packet classification*, (iv) *pattern matching* and (v) *deep packet inspection and analysis*. In order to achieve wire-speed 100 Gb throughput, the hardware architectures employ deep pipelines, prefect hashing and other techniques.

All introduced hardware architectures were implemented in the scope of Liberouter project [24] and are available as a set of acceleration cores for FPGA technology. These cores can be used in any PCIe FPGA acceleration card to accelerate network applications and devices to achieve wire-speed 100 Gbps throughput.

4.1 Results

Two hardware architectures were designed for *packet parsing and header field extraction*. The first architecture is optimized for high flexibility, because the parser can be generated from XML description of network protocols. Unfortunately, 100 Gbps throughput can be achieved on current FPGAs only at the cost of many hardware resources. Therefore, we have designed a modular low-latency architecture, which is highly optimised to FPGA logic utilization, processing speed and low latency. The results obtained by means of the modular low-latency packet parser are shown in Table 4.1.

We can see that the new low-latency modular packet parser uses only 1.19 % of the Virtex-7 870HT FPGA to achieve throughput over 100 Gbps and only 4.88 % for throughput over 400 Gb/s. The latency is below 40 ns. These results are better than previous hardware architectures [5] with FPGA logic utilization over 10 % and latency over 300 ns for 300+ Gb/s throughput. Our parser uses 68 % less FPGA resources and has 90 % smaller latency than [5] for throughput over 300 Gb/s. Moreover, the parser can be finely tuned trough the design space exploration to meet the demands of a particular application.

The hardware architecture for *longest prefix matching* employes a deep pipeline to achieve 100Gb throughput. It provides a highly efficient compression of IP prefixes to support large routing tables. IP prefixes are usually stored in the trie data structure, which is too large to be stored in the on-chip memory. Therefore, we proposed and developed an

Data Width	Pipes	Throughput [Gb/s]	Latency [ns]	LUT-FF pairs
256	0	14.5	17.1	3 238
512	0	28.4	18.0	4 053
2 048	0	96.9	21.1	17 685
2 048	1	158.5	25.9	18 547
2 048	2	212.8	28.9	18 317
2 048	4	333.0	30.8	21 775
2 048	5	352.0	34.9	22 373
2 048	7	453.0	36.2	26 728
2 048	8	478.1	38.6	29 301
1 024	?	325	309	67 902

Table 4.1: Throughput and latency of low-latency modular packet parser

efficient compression technique for the trie data structure. The most common shapes in the trie data structure are compressed by pre-designed instructions, which are executed in every pipeline stage. The memory efficiency is plot in Table 4.2. We can see that the proposed hardware architecture has very high memory efficiency for IPv6 protocol. Moreover, the processing is highly pipelined to achieve wire-speed 100 Gbps throughput.

Table 4.2: Memory demands of the proposed highly pipelined LPM architecture for available IP prefix sets, comparison to TBM and SST architectures.

Prefix Set	**Prefixes**	**Memory [Kb]**	**Savings**	
IPv4		**New Nodes**	**TBM (*SL=5*)**	**SST (*K=32*)**
rrc00	332 118	6 330.800	34.663 %	8.652 %
IPv4-space	220 779	3 571.360	37.367 %	12.488 %
route-views	442 748	7 779.840	34.853 %	11.340 %
IPv6		**New Nodes**	**TBM (*SL=3*)**	**SST (*K=32*)**
AS1221	10 518	475.760	55.822 %	19.159 %
AS6447	10 814	493.840	56.107 %	19.977 %
Generated IPv6		**New Nodes**	**TBM (*SL=4*)**	**SST**
rrc00_ipv6	319 998	21 264.320	75.630 %	N/A
IPv4-space_ipv6	150 157	10 412.160	76.314 %	N/A
route-views_ipv6	439 880	29 039.520	75.574 %	N/A

To achieve 100 Gbps throughput for the *packet classification* we developed a new hardware architecture, which uses a perfect hash function to match the classification rule in a single processing step. If the packet classification is defined on multiple dimensions (IP addresses, Port number, etc.) and rule matching is done in a single step, the architecture has to deal with crossproduct-rules. Therefore, we have modified the perfect hash function to map all crossproduct-rules to the appropriate rule number. It means that crossproduct-rules are stored only in the hash table. This optimisation significantly reduces memory requirements and increases processing speed, because our target commercial application requires only two accesses to the off-chip memory.

The performance results are shown in Table 4.3. We can see that introduced Perfect Hash Crossproduct Algorithm (PHCA) is able to process 150 millions packets per seconds with only one QDR SRAM. It means that wire-speed 100 Gb throughput has been

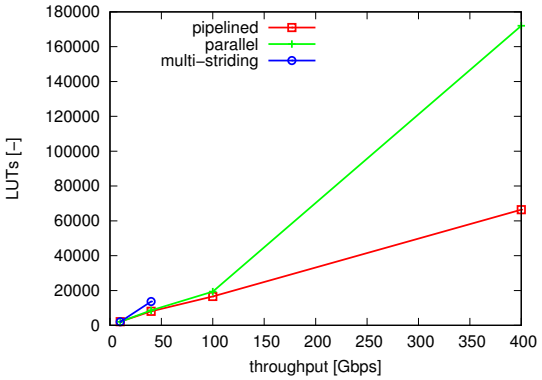

Figure 4.1: LUT occupation vs throughput for L7 great

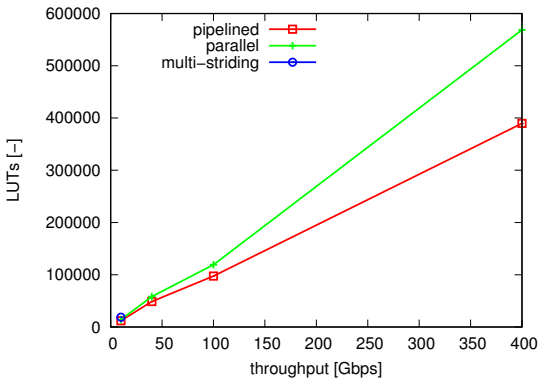

Figure 4.2: LUT occupation vs throughput for Snort backdoor rules

achieved. The DCFL algorithm has low memory requirements, but need many memory accesses. Thus wire-speed 100 Gbps throughput can be achieved only with many on-chip memory blocks [36]. Unfortunately, the on-chip memory is expensive and too small for large classification rulesets.

The architectures with Bloom filters need multiple accesses to the off-chip memory and have much lower throughput than PHCA. Memory requirements are compared in Table 4.4. Bloom filters are represented by MSCA algorithm introduced by John Lockwood. We can see that both hardware architectures (MSCA and PHCA) have comparable memory requirements and for all available rulesets, both architectures need only one QDR SRAM to store all data, but PHCA has achieved wire-speed 100 Gbps throughput.

Table 4.3: Throughput (in millions packets per second) for several data bus widths of 300MHz QDR memory.

Data Width	**Bloom Filters** 4	6	8	**PHCA**
9	9.38	6.25	4.688	150
18	18.75	12.50	9.375	150
36	37.50	25.50	18.750	150
72	75.00	50.50	37.500	150

Table 4.4: Memory size of the Rule Search Stage (kbits).

Ruleset	**DCFL**	**MSCA**	**PHCA**
synth1	15.7	1020.6	251.5
synth2	25.1	1921.6	415.7
rules1	11.1	958.5	11028.5
rules2	14.8	34.4	3266.6
rules3	30.3	92.7	346.9
rules4	93.7	368.5	599.2

The *pattern matching* is a time-critical operation, which is used in many network security systems. As multi-striding and spatial stacking techniques are not able to scale the pattern matching throughput over 40 Gbps, we proposed a new hardware architecture, which uses multiple pipelined finite state machines (FSMs). As the architecture has only local interconnections, the frequency is not affected by the number of FSMs and the processing speed can scale up to hundreds Gbps. The scalability of the architecture is shown in Figure 4.1 for L7 Great ruleset and in Figure 4.2 for Snort backdoor ruleset.

For both sets of regular expressions, the multi-striding (blue line) can be utilized to increase the throughput up to 40 Gbps. Synthesis tools were not able to meet our constrains and create a valid FPGA bitstream for higher throughput. We can see in both Figures (4.1 and 4.2) that Parallel and Pipelined Automata can scale the throughput to hundreds of Gbps, but proposed Pipelined automata have significantly lower FPGA logic utilization.

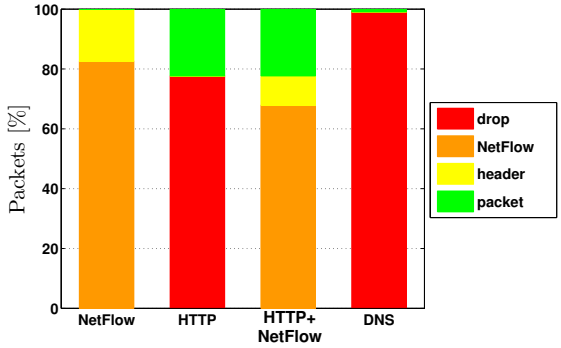

Figure 4.3: Portions of hardware pre-processing types in tested use cases

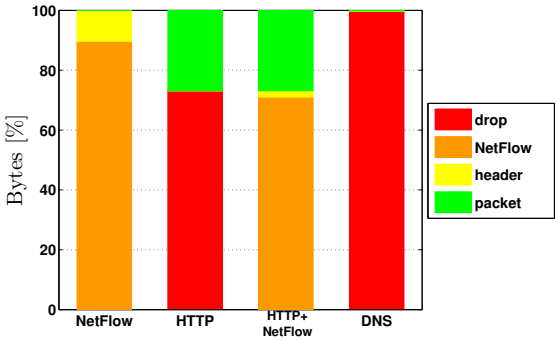

Figure 4.4: Portions of hardware pre-processing types in tested use cases

Network security and monitoring applications need flexible *deep packet inspection* and precise analysis of application protocols (L7 analysis). Even if regular expression matching is used in many network security systems, new security threats require more powerful and more flexible analysis. Therefore, we introduced a new Software Defined Monitoring (SDM) concept, which provides a flexible hardware acceleration of L7 analysis. The concept uses smart monitoring tasks implemented in the software and configurable hardware accelerator. The monitoring tasks reside in the software and can easily control the level of processing in the hardware and the level of information provided by the hardware accelerator for further software processing.

The SDM concept has been tested in four realistic use cases: (i) *measurement of NetFlow statistics*, (ii) *analysis of HTTP protocol*, (iii) *measurement of Netflow statistics and analysis of HTTP protocol* , (iv) *DNS security analysis*. Figures 4.3 and 4.3 show the portions of all packets and bytes pre-processed in the hardware. These hardware pre-processing utilizations lead to a reduction of software application load. The portion of packets processed in software is marked by green color. The yellow color corresponds to the pre-processing of packets in hardware and remaining colors (red and orange) represent the packets, which are completely processed in hardware.

4.2 Deployment and Usage

We have designed COMBO-CG [23] acceleration card as a hardware platform for 100 Gbps network security and monitoring applications. The card is shown in Figure 4.5. It has one 100GbE interface, powerful Virtex-7 FPGA, three QDR SRAM memories and fast PCIe interface. The COMBO-CG card was the first 100 Gbps FPGA card, which was able to utilize bifurcation technology (the connection of two PCI Express gen.3 x8 blocks through one x16 physical interface) to capture all packets from the network to the host PC at wire-speed without any packet drop.

The core of the COMBO-CG card is unique firmware based on hardware architectures presented in this thesis (packet header parsing, header field extraction and packet classification). The architectures enable wire-speed packet processing in the FPGA even for 100 Gbps lines. Next unique feature of the card is an efficient distribution of packets among the CPU cores of the host computer based on packet header fields. The packets are distributed equally among the cores. Moreover, all packets of the same flows are sent to the same CPU core, which significantly reduces communication among CPU cores and conse-

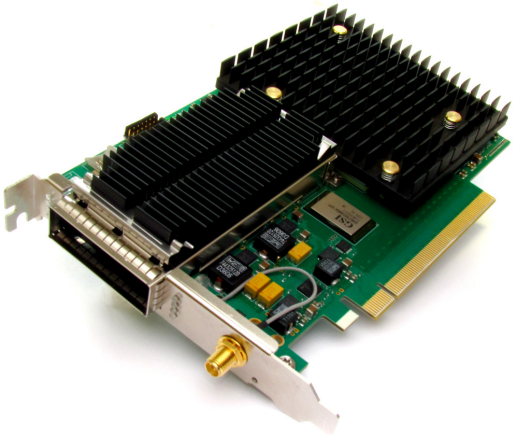

Figure 4.5: COMBO-CG hardware acceleration card.

quently increases performance of target network applications. The card together with the firmware is presented by Xilinx [66] as a recommended solution for packet capture in 100Gb networks. Xilinx is the market leading FPGA chip designer and producer.

The COMBO-CG cards were deployed into the National Research and Education Network CESNET2. All external links of CESNET2 network are monitored by probes accelerated by COMBO-CG cards. These probes provide analysis of network traffic and export NetFlow and IPFIX statistics to collectors for further analysis. The deployment of cards in CESNET2 network is shown in Figure 4.6.

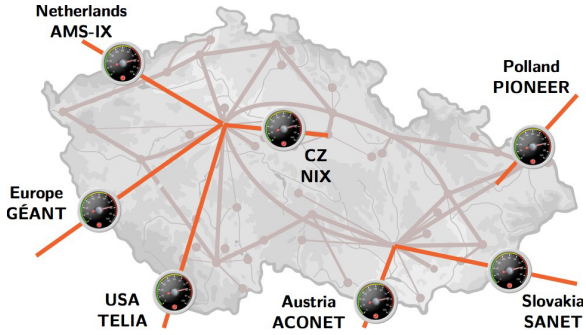

Figure 4.6: Pilot deployment of COMBO-CG acceleration cards in CESNET2 network.

The card has been created in early days of 100 Gbps Ethernet standard, which was very important for commercial utilization in all areas related to the high speed networks. Therefore, the card has been commercialized by technology transfer to Flowmon Networks and Netcope Technologies companies. These companies have sold many cards to big Internet Service Providers in Great Britain, Switzerland, Canada and other countries. Many COMBO-CG cards have been delivered also as an accelerator of lawful interception solution or as a core component of test equipment. A market leader for the network test equipment have used these cards in their network protocol testers.

The COMBO-CG card has been designed in the scope of CESNET Liberouter project [24] and DMON100 project (Distributed System for Complex Monitoring of High-Speed Networks) funded by Technology Agency of Czech Republic. I was a principal investigator of the DMON100 project. The cooperation between CESNET and Netcope Technologies

on the card has been awarded by 2nd place in the competition the Best Cooperation of the Year 2016. The objective of the competition is to increase awareness of the significance of cooperation between universities, research organisations and the private sector, and the practical impact of the transfer of research findings into practice. The competition was organised by the Association for Foreign Investment and the American Chamber of Commerce in the Czech Republic.

The card succeeded also in the 15th edition of the prestigious annual Czech Head competition. CESNET and Netcope Technologies acquired the Industrie Prize, awarded by the Ministry of Industry for the most innovative technologies. The Czech Head competition is annually organised by the eponymous company, along with the Office of the Czech Government. It is one of the most prestigious national award for science and research in the Czech republic. Laureates of the Czech Head awards for science, research and innovation have been handed out in several categories since 2002.

4.3 Conclusions

This thesis introduced hardware architectures for 100 Gbps network packet processing using FPGA technology. The architectures provide hardware acceleration of the most frequently used time-critical operations in computer networks and are used in COMBO-CG acceleration card, which has been developed as a hardware platform for 100 Gbps network applications and devices.

The papers included into this thesis are focused on five time-critical operations:

Packet parsing – two architectures have been presented in this thesis. The first flexible hardware architecture is generated from XML description. Extracted header fields can be changed on runtime without FPGA reconfiguration. This architecture was presented at IEEE DDECS'09 conference and a corresponding *paper I* attracted 13 citations. The second modular low-latency architecture is optimized to achieve high throughput and low FPGA logic utilization. It was presented at IEEE DDECS'14 conference and a corresponding *paper II* attracted 2 citations.

Longest prefix matching – presented architecture employs a deep pipeline to achieve wire-speed 100 Gb throughput. Moreover, memory requirements are reduced by compression of the trie data structure. The architecture was presented at IEEE FPL'13 conference and a corresponding *paper III* attracted 2 citations.

Packet classification – in order to achieve wire-speed 100 Gbps throughput, we have designed a hardware architecture, which utilizes perfect hash function to look-up classification rule in a single step. The architecture was presented at IEEE/ACM FPGA'09 conference and a corresponding *paper IV* attracted 16 citations.

Pattern matching – as the current pattern matching speed is limited to 40 Gbps, we designed a hardware architecture with Pipelined Automata to scale the throughput to hundreds of Gbps. The architecture was presented at IEEE FPT'16 conference and was patented under the Czech patent no. 306871. An application for a US patent has been submitted.

Deep packet inspection – the Software Defined Monitoring (SDM) concept was introduced at IEEE INFOCOM'14 conference. The original paper was extended by new

results and presented also in IEEE Transactions on Computers (TC) journal with impact factor 1.723. The IEEE INFOCOM *paper VI* has already attracted 11 citations.

These architectures are utilized as acceleration cores for the COMBO-CG card, which was created in early days of 100 Gbps Ethernet standard. Therefore, the card together with proposed architectures is deployed to protect perimeter of CESNET2 academic network, was successfully commercialised and received the Industrie Prize in Czech Head 2016 competition.

4.4 Future Work

The speed of network links is growing very fast. According to the Nielson law, a high-end user's connection speed grows by 50 % per year. New 400 Gbps Ethernet was ratified in December 2017 and large data centres already call for 1 Tb links. Therefore future research will be focused on packet processing for next generation of network links. It brings many challenges for packet processing. The most critical challenge is parallel processing of multiple packets within one clock cycle, because up to four packets can arrive to FPGA at once from 400 Gb link. Therefore hardware architectures for 400 Gbps packet processing have to be changed accordingly. This research is addressed by currently running Focus project [22], which is funded by Ministry of Interior of the Czech Republic. The goal of the Fokus project is to create 400 Gbps card together with hardware architectures for wire-speed packet processing.

Although FPGAs are programmable devices, it is difficult to adjust FPGA functionality to user needs. The programming in VHDL or Verilog is time consuming and too difficult for network administrators or software developers. Therefore our future research will be focused on mapping high-level languages to the FPGA technology. For custom packet processing, P4 [16] language has been introduced by Nick McKeown from Stanford university. The mapping of P4 description to the FPGA is a hot research topic, which has been partially solved by Pavel Benaček [13, 14] in the scope of Liberouter project. Unfortunately, the match/action tables and other P4 language constructions are not fully supported or optimised for high-speed processing and low resource utilization. Therefore, our future research will be focus on the optimisations of mapping P4 match/action tables to FPGA. This research will be addressed by currently running NFV200 project [21] funded by Technology Agency of the Czech Republic.

Bibliography

[1] ClearFoundation. l7-filter. Online, 2017. http://l7-filter.clearos.com/.

[2] Xilinx Inc. SDNet Development Environment: Expanding Programmability from the Control to the Data Plane. Online, 2014. Xilinx, Inc., https://www.xilinx.com/products/design-tools/software-zone/sdnet.html.

[3] Xilinx MicroBlaze Soft Processor. Xilinx, Inc., http://www.xilinx.com/tools/microblaze.htm.

[4] Xilinx Virtex–7 FPGA Family. Xilinx, Inc., http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.

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

[6] Baboescu, F.; Varghese, G.: Scalable packet classification. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM. 2001. ISBN 1-58113-411-8. pp. 199–210.

[7] Becchi, M.; Cadambi, S.: Memory-Efficient Regular Expression Search Using State Merging. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*. May 2007. ISSN 0743-166X. pp. 1064–1072.

[8] Becchi, M.; Crowley, P.: A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proceedings of the 2007 ACM CoNEXT Conference*. CoNEXT '07. New York, NY, USA: ACM. 2007. ISBN 978-1-59593-770-4. pp. 1:1–1:12.

[9] Becchi, M.; Crowley, P.: Efficient Regular Expression Evaluation: Theory to Practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '08. New York, NY, USA: ACM. 2008. ISBN 978-1-60558-346-4. pp. 50–59.

[10] Becchi, M.; Crowley, P.: Extending Finite Automata to Efficiently Match Perl-compatible Regular Expressions. In *Proceedings of the 2008 ACM CoNEXT Conference*. CoNEXT '08. New York, NY, USA: ACM. 2008. ISBN 978-1-60558-210-8. pp. 25:1–25:12.

[11] Becchi, M.; Crowley, P.: A-DFA: A Time- and Space-Efficient DFA Compression Algorithm for Fast Regular Expression Evaluation. *ACM Trans. Archit. Code Optim.*. vol. 10, no. 1. April 2013: pp. 4:1–4:26. ISSN 1544-3566.

[12] Becchi, M.; Wiseman, C.; Crowley, P.: Evaluating Regular Expression Matching Engines on Network and General Purpose Processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '09. New York, NY, USA: ACM. 2009. ISBN 978-1-60558-630-4. pp. 30–39.

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

[14] Benáček, P.; Puš, V.; Kubátová, H.; et al.: P4-To-VHDL: Automatic generation of high-speed input and output network blocks. *Microprocessors and Microsystems*. vol. 56. 2018: pp. 22 – 33. ISSN 0141-9331.

[15] Bloom, B.: Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*. 1970.

[16] Bosshart, P.; Daly, D.; Gibb, G.; et al.: P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*. vol. 44, no. 3. July 2014: pp. 87–95. ISSN 0146-4833.

[17] Braun, F.; Lockwood, J.; Waldvogel, M.: Protocol wrappers for layered network packet processing in reconfigurable hardware. *Micro, IEEE*. vol. 22, no. 1. 2002: pp. 66–74. ISSN 0272-1732.

[18] Brodie, B. C.; Taylor, D. E.; Cytron, R. K.: A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In *33rd International Symposium on Computer Architecture (ISCA'06)*. 2006. ISSN 1063-6897. pp. 191–202.

[19] Caswell, B.; Foster, J. C.; Russell, R.; et al.: *Snort 2.0 Intrusion Detection*. Syngress Publishing. 2003. ISBN 1931836744.

[20] Caulfield, A.; Chung, E.; Putnam, A.; et al.: A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. October 2016.

[21] CESNET, z.s.p.o.: Platform for Acceleration of Network Functions Virtualization, research project funded by TAČR. 2017-2019.

[22] CESNET, z.s.p.o.: Adaptive Management of Data Collection and Analysis in High-Speed Networks (FOKUS), research project funded by MV ČR. 2017-2020.

[23] CESNET, z.s.p.o.: COMBO-CG card. 2018.
Retrieved from: https://www.liberouter.org/combo-100g/

[24] CESNET, z.s.p.o.: Liberouter project. 2018.
Retrieved from: https://www.liberouter.org/

[25] Clark, C. R.; Schimmel, D. E.: Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *In Proceedings of 13th International Conference on Field Program.* 2003. pp. 956–959.

[26] Clark, C. R.; Schimmel, D. E.: Scalable pattern matching for high speed networks. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*. April 2004. pp. 249–257.

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

[28] Deri, L.; Martinelli, M.; Bujlow, T.; et al.: nDPI: Open-source high-speed deep packet inspection. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*. Aug 2014. ISSN 2376-6492. pp. 617–622.

[29] Dharmapurikar, S.; Song, H.; Turner, J.; et al.: Fast packet classification using Bloom filters. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. New York, NY, USA: ACM. 2006. ISBN 1-59593-580-0. pp. 61–70.

[30] Eatherton, W.; Varghese, G.; Dittia, Z.: Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *SIGCOMM Comput. Commun. Rev.*. vol. 34, no. 2. April 2004: pp. 97–122. ISSN 0146-4833.

[31] Emmerich, P.; Raumer, D.; Wohlfart, F.; et al.: Performance characteristics of virtual switching. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. Oct 2014. pp. 120–125.

[32] Gupta, P.; McKeown, N.: Packet classification using hierarchical intelligent cuttings. In *Proc. Hot Interconnects*. 1999.

[33] Gupta, P.; Prabhakar, B.; Boyd, S. P.: Near Optimal Routing Lookups with Bounded Worst Case Performance. In *INFOCOM*. 2000. pp. 1184–1192.

[34] IEEE 802.3 Ethernet Working Group: IEEE 802.3 industry connections ethernet bandwidth assessment. Technical report,. 2012.

[35] Kaneta, Y.; Yoshizawa, S.; i. Minato, S.; et al.: Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching. In *Field-Programmable Technology (FPT), 2010 International Conference on*. Dec 2010. pp. 21–28.

[36] Kekely, M.; Korenek, J.: Mapping of P4 match action tables to FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. Sept 2017. pp. 1–2.

[37] Kobierský, P.; Kořenek, J.; Polčák, L.: Packet header analysis and field extraction for multigigabit networks. In *Proceedings of the 2009 12th International Symposium on Design and Diagnostics of Electronic Circuits&Systems*. DDECS. Washington, USA: IEEE Computer Society. 2009. ISBN 978-1-4244-3341-4. pp. 96–101.

[38] Kořenek, J.: Fast Regular Expression Matching Using FPGA. *Information Sciences and Technologies Bulletin of the ACM Slovakia*. vol. 2, no. 2. 2010: pp. 103–111. ISSN 1338-1237.

[39] Kozanitis, C.; Huber, J.; Singh, S.; et al.: Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System. In *IEEE INFOCOM*. mar. 2010. ISSN 0743-166X.

[40] Kořenek, J.; Puš, V.; Blaho, J.: Memory Optimization for Packet Classification Algorithms. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. Association for Computing Machinery. Association for Computing Machinery. 2009. ISBN 978-1-60558-630-4. pp. 165–166.

[41] Kumar, S.; Chandrasekaran, B.; Turner, J.; et al.: Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. ANCS '07. New York, NY, USA: ACM. 2007. ISBN 978-1-59593-945-6. pp. 155–164.

[42] Kumar, S.; Dharmapurikar, S.; Yu, F.; et al.: Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '06. New York, NY, USA: ACM. 2006. ISBN 1-59593-308-5. pp. 339–350.

[43] Lakshman, T. V.; Stiliadis, D.: High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.*. vol. 28, no. 4. 1998: pp. 203–214. ISSN 0146-4833.

[44] Le, H.; Prasanna, V. K.: Scalable Tree-based Architectures for IPv4/v6 Lookup Using Prefix Partitioning. vol. 61, no. 7. July 2012: pp. 1026–1039. ISSN 0018-9340.

[45] Lee, H.; Jiang, W.; Prasanna, V. K.: Scalable High-Throughput SRAM-Based Architecture for IP Lookup Using FPGA. In *FPL '08*. IEEE. 2008.

[46] Lin, C.; Huang, C.; Jiang, C.; et al.: Optimization of Pattern Matching Circuits for Regular Expression on FPGA. *IEEE Trans. VLSI Syst.*. vol. 15, no. 12. 2007: pp. 1303–1310.

[47] Matoušek, J.; Skačan, M.; Kořenek, J.: Memory efficient IP lookup in 100 GBPS networks. In *2013 23rd International Conference on Field programmable Logic and Applications*. Sept 2013. ISSN 1946-147X. pp. 1–8.

[48] Netronome Inc.: Hardware Acceleration for Network Services. 2018. Retrieved from: https://www.netronome.com/media/documents/WP_Hardware_Acceleration.pdf

[49] NXP Inc.: QorIQ LS2088A processor. 2018. Retrieved from: https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/qoriq-layerscape-arm-processors/qoriq-layerscape-2088a-and-2048a-multicore-communications-processors:LS2088A

[50] Ovtcharov, K.; Ruwase, O.; Kim, J.-Y.; et al.: Toward Accelerating Deep Learning at Scale Using Specialized Hardware in the Datacenter. In *Proceedings of the 27th IEEE HotChips Symposium on High-Performance Chips (HotChips 2015)*. IEEE. August 2015.

[51] Paxson, V.: Bro: A System for Detecting Network Intruders in Real-time. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. SSYM'98. Berkeley, CA, USA: USENIX Association. 1998. pp. 3–3.

[52] Pfaff, B.; Pettit, J.; Koponen, T.; et al.: The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Berkeley, CA, USA: USENIX Association. 2015. ISBN 978-1-931971-218. pp. 117–130.

[53] Putnam, A.; Caulfield, A.; Chung, E.; et al.: A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*. IEEE Press. June 2014. ISBN 978-1-4799-4394-4. pp. 13–24.

[54] Puš, V.; Kekely, L.; Kořenek, J.: Design methodology of configurable high performance packet parser for FPGA. In *Design and Diagnostics of Electronic Circuits Systems, 17th International Symposium on*. April 2014. pp. 189–194.

[55] Puš, V.; Kořenek, J.: Fast and Scalable Packet Classification Using Perfect Hash Functions. In *FPGA '09: Proceedings of the 17th international ACM/SIGDA symposium on Field programmable gate arrays*. New York, NY, USA: ACM. 2009.

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

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

[58] Ruiz-Sánchez, M. A.; Biersack, E. W.; Dabbous, W.: Survey and Taxonomy of IP Address Lookup Algorithms. vol. 15, no. 2. March 2001: pp. 8–23. ISSN 0890-8044.

[59] Sidhu, R.; Prasanna, V. K.: Fast Regular Expression Matching Using FPGAs. In *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. FCCM '01. Washington, DC, USA: IEEE Computer Society. 2001. ISBN 0-7695-2667-5. pp. 227–238.

[60] Singh, S.; Baboescu, F.; Varghese, G.; et al.: Packet classification using multidimensional cutting. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM. 2003. ISBN 1-58113-735-4. pp. 213–224.

[61] Song, H.; Turner, J.; Lockwood, J.: Shape Shifting Tries for Faster IP Route Lookup. In *Proc. of the 13th IEEE International Conference on Network Protocols (ICNP'05)*. IEEE Computer Society. 2005. pp. 358–367. ISBN 0-7695-2437-0.

[62] Sourdis, I.; Bispo, J.; Cardoso, J. M. P.; et al.: Regular Expression Matching in Reconfigurable Hardware. *Journal of Signal Processing Systems*. vol. 51, no. 1. 2008: pp. 99–121. ISSN 1939-8115.

[63] Srinivasan, V.; Varghese, G.; Suri, S.; et al.: Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*. vol. 28, no. 4. 1998: pp. 191–202. ISSN 0146-4833.

[64] Taylor, D.; Turner, J.: Scalable Packet Classification using Distributed Crossproducing of Field Labels. In *IEEE INFOCOM 2005, 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*. July 2005. pp. 269–280.

[65] Weaver, N.; Paxson, V.; Gonzalez, J. M.: The Shunt: An FPGA-based Accelerator for Network Intrusion Prevention. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*. FPGA '07. New York, NY, USA: ACM. 2007. ISBN 978-1-59593-600-4. pp. 199–206.

[66] Xilinx Inc.: FPGA producer. 2018.
Retrieved from: http://www.xilinx.com/

[67] Yang, Y. H.; Prasanna, V.: High-Performance and Compact Architecture for Regular Expression Matching on FPGA. *IEEE Transactions on Computers*. vol. 61, no. 7. July 2012: pp. 1013–1025. ISSN 0018-9340.

[68] Yang, Y.-H. E.; Jiang, W.; Prasanna, V. K.: Compact Architecture for High-throughput Regular Expression Matching on FPGA. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '08. New York, NY, USA: ACM. 2008. ISBN 978-1-60558-346-4. pp. 30–39.

[69] Yu, F.; Chen, Z.; Diao, Y.; et al.: Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. ANCS '06. New York, NY, USA: ACM. 2006. ISBN 1-59593-580-0. pp. 93–102.

[70] Yun, S.; Lee, K.: Optimization of Regular Expression Pattern Matching Circuit Using At-Most Two-Hot Encoding on FPGA. In *2010 International Conference on Field Programmable Logic and Applications*. Aug 2010. ISSN 1946-147X. pp. 40–43.

Appendix A

Included Papers

A.1 Paper I

Packet Header Analysis and Field Extraction for Multigigabit Networks

Packet Header Analysis and Field Extraction for Multigigabit Networks

Petr Kobierský
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66,
Brno, Czech Republic
Email: ikobier@fit.vutbr.cz

Jan Kořenek
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66,
Brno, Czech Republic
Email: korenek@fit.vutbr.cz

Libor Polčák
CESNET z. s. p. o.
Zikova 4, 160 00,
Prague, Czech Republic
Email: polcak_l@liberouter.org

***Abstract*—Packet header analysis and extraction of header fields needs to be performed in all network devices. As network speed is increasing quickly, high speed packet header processing is required. We propose a new architecture of packet header analysis and fields extraction intended for high-speed FPGA-based network applications. The architecture is able to process 20 Gbps network links with less than 12 percent of available resources of Virtex 5 110 FPGA. Moreover, the presented solution can balance between network throughput and consumed hardware resources to fit application needs. The architecture for packet header processing is generated from standard XML protocol scheme and is strongly optimised for resource consumption and speed by an automatic HDL code generator. Our solution also enables to change the set of extracted header fields on-line without FPGA reconfiguration.**

I. Introduction

In the recent decade a rapid development in area of network technologies can be observed. The increase in the link capacity from 100 Mbps to 10 Gbps and more puts higher demands on devices that have to process such network traffic. Building efficient, robust and secure networks requires several types of network devices like routers, firewalls, monitoring probes, Intrusion Detection Systems, etc. Historically, these devices were implemented using conventional computers, but this approach is insufficient for current high-speed networks, therefore expensive hardware devices are used for this purpose (Cisco, Juniper, etc.).

Each of network device works at least with packet headers; it processes them and extracts information required by higher level blocks, e.g. a routing algorithm. This task is usually performed in hardware network devices by network processors (NP) [1] which are able to achieve throughput from 10 to 100 Gbps.

Unfortunately, NPs lack a good throughput characteristic for such tasks as regular expression matching [2], [3] (required by IDS systems) and packet classification [4], [5]. NP often interfaces with custom ASIC/FPGA hardware designed for these specific purpose in order to solve this issue.

Multiple chip solution can not be used in many embedded devices because of cost, size or power consumption. To reduce these factors a single chip application (SoC) [6] is required. In recent years, the FPGA technology is most often used for designing and prototyping of a network domain SoC. In comparison with the ASIC technology, the FPGA technology reduces development costs and offers the possibility of reconfiguration. Especially changing hardware behaviour during product life cycle enables a long lifetime of network devices, which yields infrastructure cost savings.

Despite the importance of protocol analysis and header extraction for FPGA technology, this issue is currently omitted. Only a small number of papers are focused on architectures suitable for FPGA chips and high-speed networks.

A library of layered protocol processing engines [7] has been introduced by John Lockwood in FPX platform. Each engine is a hardware block responsible for processing of one protocol from a single ISO/OSI layer. The engines have a well-defined interface for outer and inner protocol which enables nesting several engines together. As a result, it is possible to achieve a modular design which can support applications for different protocols and level of abstraction.

Another approach, how to perform protocol analysis and extraction is a usage of a processor IP core. The Liberouter [8] project uses a custom-made 16-bit RISC processor designed to analyse protocols and extract selected fields on L2–L4 ISO/OSI layers. This custom processor is able to achieve throughput up to 900 Mbps on Virtex II FPGA.

A processing engine [9] in form of a state machine described in high-level abstraction language (Handel-C) was later proposed. The state machine achieves higher throughputs than the custom processor and consumes almost the same number of hardware resources, but still cannot be simply used for 10 Gbps networks.

Moreover, all introduced architectures suffer from problems with adaptation to new network protocols. If a new protocol specification is created and needs to be supported, an experienced designer familiar with HDL language or processor architecture is required. Therefore, the interval between new release of the protocol specification and its support in hardware device is very long. Hard-wired approaches are also not suitable for monitoring and security network applications (e.g. flexible flow monitoring [10]), where the set of extracted header fields needs to be changed on-line.

Another problem of available architectures is a low through-

978-1-4244-3339-1/09/$25.00 ©2009 IEEE

put which limits their usage to 1 Gbps networks only. Higher throughput can be achieved only by parallel usage of several processing units which causes significant overhead in FPGA logic resources given by multiple engines and packet distribution logic.

In order to challenge the presented issues, we propose an architecture of packet header field extractor (HFE) which is required in all network devices. Our solution can balance between network throughput and consumed hardware resources. We can dynamically react to new network protocols thanks to automatic conversion from protocol description in XML format. Using this abstract description and automatic HDL code generator, we are able to create an efficient and configurable processing engine with very small consumption of hardware resources.

This paper is organized as follows: Section II introduces an hardware architecture for packet header analysis and field extraction. Section III shows an XML schema used for describing network protocols description and explains its transformation to HDL code and HFE microcode. The evaluation of proposed architecture is described in Section IV. The results are summarized in Section V.

II. Architecture

A new architecture of HFE engine intended for the FPGA technology is introduced in this section. The proposed architecture described in Fig. 1 is composed of two modules responsible for protocol analysis and header field extraction. This separation is important because the parts of the problem require different hardware architecture.

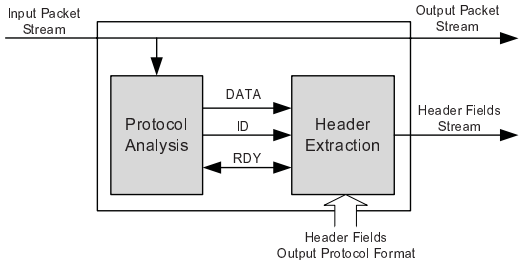

Fig. 1. Block level diagram of HFE

The proposed architecture communicates with other processing blocks in target application using Xilinx Local-Link [11] protocol. This protocol has been chosen because of configurable data width and its easy implementation in potentially interfacing IP cores. By increasing the data width of the Local-Link protocol and configuring HFE engine to be able to process an input word of such width in a single clock cycle, we are able to achieve sufficient throughput even for 10 Gbps networks.

One of the core parts of HFE is a *protocol analysis module*, which is required for identification of protocols included in the processed packet. The module reads words from input and outputs an identifier based on their content. The identifier is a unique number which is generated for every possible combination of supported protocols header fields in the input word (it is actually an encoded information about which protocol header fields on which byte positions are contained in the input word).

The protocol analysis module is realized by a Mealy machine and several internal registers with associated combinational logic. The machine is constructed automatically from provided XML description of network protocols. The process of analysis module generation is described in Section III.

The second block of the HFE architecture is responsible for extraction of selected header fields from the input word and positioning them to correct offset in the output data frame. The *header extraction module* is controlled by a microcode stored in an on-chip memory. The microcode is generated from the XML description of the output frame, which will be further described in Section III. The proposed architecture of the extraction engine is shown in Fig. 2.

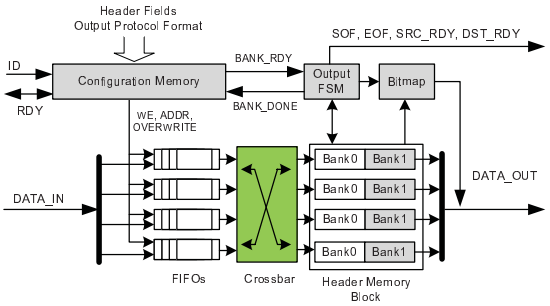

Fig. 2. Extraction Engine

The extraction module determines for each byte of the input word whether it has to be extracted or discarded. This information is read from a *configuration memory* based on the FSM transition identifier passed together with the input data word from the protocol analysis module. Bytes which were marked for extraction are added to corresponding FIFO together with their position in the output stream.

The position parameter is used as a switching address for a *crossbar*. The extracted bytes switched and stored in correct position in *header memory block* are based on this address. This block is composed of 8-bit dual-port memories which are logically divided into several banks working like a circle buffer.

Extracted header fields from currently processed packet are stored into empty bank while the *output FSM* can send extracted data from previous packet to HFE output. After all packet header fields are processed, the bank is marked as empty and can be used to store extracted bytes from the next packet. This store and forward architecture enables to have a configurable output format independent of the original position of the header fields in the input packet.

The extraction module also creates a *bitmap* of valid header fields (bytes). This is because all header fields are not in every

packet e.g. TCP or UDP ports are not in ARP packets. The bitmap can be sent to HFE output with extracted header fields based on the extraction module configuration.

The proposed architecture can also deal with nested protocols (VLANs, MPLS, IP over IP tunnels). For these protocols, it is usually required to extract the most nested field or the first instance of specific field depending on the target application. This is realized by reading single bit from the configuration memory, which enables/disables overwriting of already extracted bytes.

III. HFE Generation and Configuration

In this section the process of HFE generation and configuration based on the target application requirements is described. The generation process has been designed for simple usage by a user without any knowledge of hardware design. We proposed XML-based abstract description, which is used to specify following information to custom core generator:

- network protocols description,
- protocol header fields which will be extracted,
- extracted header fields position in output stream,
- width of the input packet stream.

The XML format has been selected because of its suitability for description of such structured information. XML files can be modified by any text editor and most of the current programming languages have support for parsing documents in the XML format.

The HFE engine generation process is divided into two steps as can be seen in Fig. 3. In the first step the VHDL files which describe the protocol analysis block are created. The generated sources are synthesizable and can be directly used in several network applications.

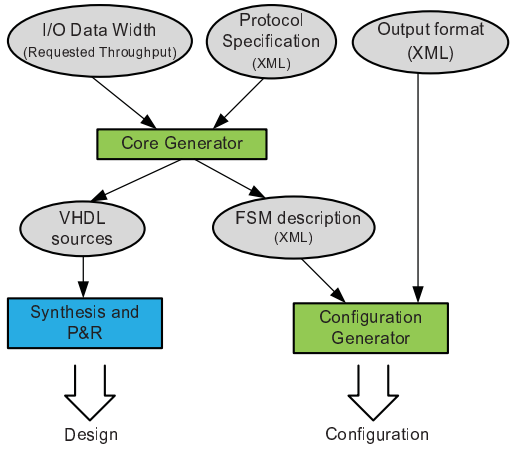

Fig. 3. Processing engine generation and configuration process

A configuration (microcode) for the extraction engine is generated in the second step. The generated microcode needs to be written into the extraction engine configuration memory. This can be done by initializing the memory during synthesis process or at runtime through a configuration bus. This means that the HFE output format can be changed not only during the synthesis process but also at runtime.

A. Transforming XML Protocol Description into VHDL

In this section the protocol analysis engine generation from XML protocol schema is described in detail. Before we describe the transformation process, the XML schema used for protocol description needs to be defined. This schema was designed to be very simple and intuitive with the ability to express all L2–L4 ISO/OSI protocols.

Using our XML schema, it is possible to define a protocol by enumerating its header fields with their sizes. Relations between protocols on different levels of ISO/OSI model is described using a jump construct, which chooses a higher level protocol based on the value of user defined variables. The content of these variables is modified by operations that can be assigned to protocol header fields. The typical operations for describing TCP/IP protocols are saving the content of a header field into a variable and decrementing a variable by a constant or the content of a header field.

In Fig. 4 an example of the Ethernet protocol description in XML is shown. It can be seen that the content of Length/Type field is stored into an internal variable which is used for selecting next protocol in the *jump* XML tag. In this particular description only the IPv4 protocol is supported.

```
<protocol name="Ethernet" id="Eth">
 <variables>
   <variable name="Next protocol" id="eth_next_proto"
             size="16" vartype="int" />
 </variables>

 <field name="Destination MAC address" id="eth_dst_mac"
        size="48" />
 <field name="Source MAC address" id="eth_src_mac"
        size="48" />
 <field name="Length/Type" id="eth_len_type"
        size="16">
   <fieldop ref="op_assign">
     <fieldopparam name="target" ref="eth_next_proto" />
     <fieldopparam name="value" ref="eth_len_type" />
   </fieldop>
 </field>

 <jump selectby="eth_next_proto">
   <case equal="0x0800" ref="IPv4" />
   <others ref="UNSUPORTED" />
 </jump>
</protocol>
```

Fig. 4. XML description of simplified Ethernet protocol

From the XML protocol description the *protocol analysis FSM* is generated. This formal model (see Definition 1) is based on the Mealy FSM, which is extended by a set of variables, where the semantic information can be stored. Thus the FSM is able to change its behavior according to the semantic information available in the packets. It is possible to perform simple operations with the variables like addition, comparison, etc. during FSM transitions.

Definition 1. *A Protocol Analysis FSM is an 8-tuple* $(S, V, \Sigma, \Lambda, \Delta, \delta, S_0, V_0)$*, where:*

- S *is a finite set of states,*

- V *is the finite set of internal variable names,*
- Σ *is the input alphabet,*
- Λ *is the output alphabet,*
- Δ *is the internal alphabet,*
- S_0 *is the initial state which is an element of* S *(*$S_0 \in S$*),*
- V_0 *is a finite set of initial values for each variable from* V *(*$V_0 \in \Delta^{|V|}$*),*
- $\delta : S \times \Delta^{|V|} \times \Sigma \rightarrow S \times \Delta^{|V|} \times \Lambda$ *is a transition function which selects the next state, updates internal variables and selects the output symbol based on the input, current state and the content of internal variables.*

In the initial phase of the VHDL generation process XML protocol descriptions are transformed into single 1-bit protocol analysis FSM. This machine accepts only a single bit of input packet stream per its transition ($\Sigma = \{0, 1\}$). It is further converted into multi-char protocol analysis FSM based on the requested throughput (the input data width parameter).

The conversion to multi-char FSM is based on an incremental search of a transition path of length n from the initial state S_0 to states from set S. Every found path is replaced by a single multi-char FSM transition which is created by merging the 1-bit transitions along the path. The set of all target states of newly created transitions that have not been processed yet is used as a set of initial states in the next step of the path search.

The multi-char FSM is then transformed into a VHDL FSM description. The next state of the FSM is selected based on internal registers and the input word. The FSM controls assignment of computed values to internal registers and outputs the identifier which is used for extraction purposes as will be described in the next section.

B. Generation of Configuration for Header Extraction Engine

Another task of the HFE generation process is the compilation of microcode for the extraction module. The main input to this process is description of requested HFE output frame in an XML document. An example of the XML output frame description is shown in Fig. 5.

This example contains several fields which will be filled with the content of protocol header fields based on the description inside *content* tag. It can be seen that it is possible to assign different protocol header fields to a single output field (in this example UDP and TCP ports), which is useful for packet classification applications. The *extract* attribute enables to select whether the first or the last occurrence of the header field is extracted. This feature is important when processing nested protocols IP-over-IP, nested VLANs, etc.

The output frame format and information about possible header fields combinations in the input word including their assigned identifier are directly used for generation of content of the extraction engine configuration memory. A single control word is generated for each transition. This word contains command for each input byte which controls:

- byte extraction,
- position in the output stream,

```
<extract>

<output>
<field id="src_mac" length="6" />
<field id="dst_mac" length="6" />
<field id="src_ip" length="4" />
<field id="dst_ip" length="4" />
<field id="protocol" length="1" />
<field id="src_port" length="2" />
<field id="dst_port" length="2" />
</output>

<content>
<extract fieldref="eth_src_mac" outref="src_mac"
            extract="first" />
<extract fieldref="eth_dst_mac" outref="dst_mac"
            extract="first" />
<extract fieldref="ipv4_src_addr" outref="src_ip"/>
<extract fieldref="ipv4_dst_addr" outref="dst_ip"/>
<extract fieldref="ipv4_protocol" outref="protocol"/>
<extract fieldref="tcp_src_port" outref="src_port"/>
<extract fieldref="tcp_dst_port" outref="dst_port"/>
<extract fieldref="udp_src_port" outref="src_port"/>
<extract fieldref="udp_dst_port" outref="dst_port"/>
</content>

</extract>
```

Fig. 5. XML description of the HFE output frame

- overwriting of already extracted value (nested protocols).

IV. Results

The proposed architecture was evaluated on Xilinx Virtex 5 FPGA. We have defined four protocol sets in XML language and used them for proposed HFE engine evaluation. All prepared protocol descriptions are mentioned in Table I. Using the core generator, several HFE engines with different number of supported protocols and the input word data width were generated. As can be seen in Table II and Table III, number of supported protocols and protocol types have only limited influence to the consumed FPGA resources and processing speed.

TABLE I
Evaluation set description

Protocol Name	Basic	VLAN	MPLS	IPV6
Ethernet	X	X	X	X
VLAN		X	X	X
MPLS			X	X
IPv4	X	X	X	X
IPv6				X
TCP	X	X	X	X
UDP	X	X	X	X

We have performed the Synthesis and Place & Route for all generated designs using Xilinx ISE tool. The maximum frequency and consumed hardware resources were obtained from P&R results. Table II shows the HFE throughput for its different configurations. The throughput was computed from the input width and maximum design frequency. For the 128-bit configuration the target throughput was over 15 Gbps, which is more than sufficient for processing of 10 Gbps network lines even for the shortest packets. The frequency parameter decreases from 165 MHz to 115 MHz with the

increase of the input data width because of long combinatorial paths in the extraction part (crossbar) of the HFE, which can be further optimized to increase frequency.

TABLE II
MAXIMUM ACHIEVED THROUGHPUT FOR DIFFERENT INPUT DATA WIDTHS

Input Width	Basic	VLAN	MPLS	IPV6
16	2.67 Gbps	2.67 Gbps	2.67 Gbps	2.67 Gbps
32	5.34 Gbps	5.34 Gbps	5.34 Gbps	5.34 Gbps
64	8.45 Gbps	8.45 Gbps	8.45 Gbps	8.45 Gbps
128	15.23 Gbps	15.23 Gbps	14.98 Gbps	15.23 Gbps

We have also evaluated the stand-alone protocol analysis module (FSM) to determine the influence of the number of supported protocol to its throughput. The results show that the maximum frequency for different protocol sets is almost the same.

The amount of consumed hardware resources for different HFE configurations is shown in Table III. For the most powerful configuration, only 1964 slices were consumed, which totals to only 11 % of all resources available on xc5vlx110t chip. Our solution is also very friendly to BlockRam resources which are often needed for implementing of large packet buffers in network devices. Relatively small consumption of FPGA resources enables to use the rest of the chip for building complicated network applications like firewalls, routers, IDS systems, etc.

TABLE III
SLICE / BLOCKRAM'S OCCUPATION ON VIRTEX 5 (XC5VLX110T) FOR DIFFERENT INPUT DATA WIDTHS

Input Width	Basic	VLAN	MPLS	IPV6
16	172 / 1	177 / 1	182 / 1	189 / 1
32	305 / 1	316 / 1	322 / 1	347 / 1
64	660 / 3	674 / 3	710 / 3	747 / 3
128	1713 / 5	1816 / 5	1837 / 5	1964 / 5

Table III also shows that the increase of occupied area with added support for more protocols is very low. IPv6 support consumes only less than 150 additional slices for 128-bit input word configuration. For lower input data width these differences are practically unnoticeable.

The higher throughput can be achieved not only by increasing of the input data width but also by flexible packet distribution between several processing units with lower throughput as can be seen in Fig. 6. We have used this principle to determine whether a single processing engine with 128-bit input data width is better than several less powerful engines.

In Table IV, there is an evaluation of four different HFE configurations which differ in the input data width and the number of processing engines. The packet distribution logic is included into consumed FPGA resources for configurations that use more than one processing block. The table shows that the best relation between the throughput and the occupied FPGA resources is for 4x 32-bit processing engines. This configuration is the best mainly because of 6-input LUTs

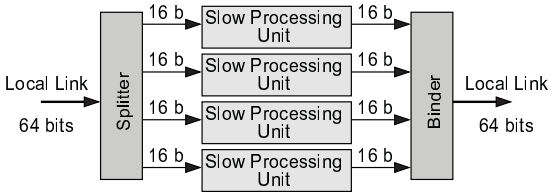

Fig. 6. Processing unit load distribution

used in Virtex 5 architecture which can be fully utilized for creating 32-bit crossbar. Another reason is that L3–L4 network protocols used during evaluation are aligned to 32 bits.

TABLE IV
DETERMINING THE OPTIMAL HFE ENGINE CONFIGURATION FOR VIRTEX 5 CHIPS

Configuration	Throughput	Area	Throughput/Area
1x 128-bit HFE	15.23 Gbps	1964 slices	7.7 Mbps/Slice
2x 64-bit HFE	16.9 Gbps	1930 slices	8.8 Mbps/Slice
4x 32-bit HFE	21.36 Gbps	2100 slices	10.2 Mbps/Slice
8x 16-bit HFE	21.36 Gbps	2825 slices	7.5 Mbps/Slice

V. CONCLUSION

In this paper, we propose a new architecture intended for packet header analysis and extraction on multigigabit networks. The presented solution has been designed especially for FPGA chips and can be used in powerful high-speed devices. The architecture is optimized for high speed processing and consumes only small amount of FPGA resources. More then 20 Gbps throughput can be achieved using less than 2000 slices of Virtex 5 FPGA. Moreover, the architecture is well designed for embedded applications, because a gigabit link can be processed by only one engine which consumes 170 slices of FPGA.

The hardware implementation of HFE engine is generated by a core generator, which enables to optimize architecture for defined parameters and balance between consumed hardware resources and network throughput. As an XML specification is used to describe network protocols, the HFE engine can be generated without the knowledge of hardware design. This significantly reduces the time required for providing support to newly issued specifications of network protocols. Moreover, due to the well designed architecture, it is possible to change the set of extracted header fields without FPGA reconfiguration. Only the internal memory has to be changed by a simple configuration data upload.

Maximum frequency of the proposed architecture decreases with growing data width. Therefore, our future work will focus on performance optimizations that enable to reach higher system frequency. We will analyse data flow dependencies to find operations which cause critical paths in the design, because these operations can be processed in advance and pipelined with packet header analysis.

ACKNOWLEDGMENT

This research has been partially supported by the Research Plan No. MSM, 6383917201 – Optical National Research Network and its New Applications and Research Plan No. MSM, 0021630528 – Security-Oriented Research in Information Technology.

REFERENCES

[1] P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk, *Network Processor Design: Issues and Practices, Volume 1*. Morgan Kaufmann, San Francisco, CA, 2003.

[2] Z. K. Baker and V. K. Prasanna, "Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs," in *Proceedings of the 14th Annual International Conference on Field-Programmable Logic and Applications (FPL '04)*, 2004.

[3] C. R. Clark and D. E. Schimmel, "Scalable Pattern Matching for High-Speed Networks," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Napa, California, 2004, pp. 249–257.

[4] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using fpga," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM Press, 2005, pp. 238–245.

[5] V. Puš and J. Kořenek, "Fast and scalable packet classification using perfect hash functions," in *FPGA '09: Proceedings of the 17th international ACM/SIGDA symposium on Field programmable gate arrays*. New York, NY, USA: ACM, 2009.

[6] R. Rajsuman, *System-on-a-Chip: Design and Test*. Artech House, Inc. Norwood, MA, USA, 2000.

[7] F. Braun, J. Lockwood, and M. Waldvogel, "Protocol wrappers for layered network packet processing in reconfigurable networks," *IEEE Micro*, vol. 22, no. 1, pp. 66–74, Jan./Feb. 2002.

[8] Liberouter, "Liberouter Project WWW Page," http://www.liberouter.org, 2006.

[9] T. Dedek, T. Marek, and T. Martínek, "High level abstraction language as an alternative to embeded processors for internet packet processing in fpga," in *2007 International Conference on Field Programmable Logic and Applications*. IEEE Computer Society, 2007, pp. 648–651.

[10] M. Žádník, J. Kořenek, O. Lengál, and P. Kobierský, "Network probe for flexible flow monitoring," in *Proc. of 2008 IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*. IEEE Computer Society, 2008, pp. 213–218.

[11] Xilinx, "LocalLink Interface Specification," July 2005.

A.2 Paper II

Design Methodology of Configurable High Performance Packet Parser for FPGA

Design Methodology of Configurable High Performance Packet Parser for FPGA

Viktor Puš, Lukáš Kekely
CESNET a. l. e.
Zikova 4, 160 00 Prague, Czech Republic
Email: pus,kekely@cesnet.cz

Jan Kořenek
IT4Innovations Centre of Excellence
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
Email: korenek@fit.vutbr.cz

***Abstract*—Packet parsing is among basic operations that are performed at all points of a network infrastructure. Modern networks impose challenging requirements on the performance and configurability of packet parsing modules. However, high-speed parsers often use a significant amount of hardware resources. We propose a novel architecture of a pipelined packet parser for FPGA, which offers low latency in addition to high throughput (over 100 Gb/s). Moreover, the latency, throughput and chip area can be finely tuned to fit the needs of a particular application. The parser is hand-optimized thanks to a direct implementation in VHDL, yet the structure is uniform and easily extensible for new protocols.**

***Keywords*—Packet Parsing; Latency; FPGA**

I. Introduction

Since computer networks evolve both in terms of speed and diversity of protocols, there is still a need for packet parsing modules at all points of the infrastructure. This is true not only in the public Internet, but also in closed, application-specific networks. There are very different expectations on packet parsers. For example, consider a multi-million dollar business of low-latency algorithmic trading. In this area, the *latency*, which has long been rather neglected parameter, suddenly becomes more important than the raw throughput. Small embedded devices, on the other hand, often require a parser to be very small (in terms of the memory and chip area), yet still to support a rather extensive set of protocols.

With a rising interest in the Software Defined Networking, it is expected that the "ossification" of networks will be on decline, and new protocols will appear at even faster rate than before. This expectation handicaps fixed ASIC parsers and favours programmable solutions: CPUs, NPUs and FPGAs. Our work focuses on FPGAs, because of their great potential in high-speed networks.

Current high-speed FPGA-based parsers can achieve a raw throughput of over 400 Gb/s at the cost of the extreme pipelining, which increases both the latency and the chip area (FPGA resources) significantly [1]. Also, the configurability issue is solved only partially. Configuring a set of supported protocols is often addressed by a higher-level protocol description followed by an automatic code generation, but the configuration of the implementation details is left unnoticed.

This paper not only presents a novel packet parser design, but also motivates engineers to create a parametrized solutions, demonstrates the need for a thorough exploration of the space of the solutions and suggests several capabilities that a High-Level Synthesis system should possess to succeed in this area.

The paper is organized as follows: Section II introduces several prior published works in this area, Section III describes our implementation of a modular parser design, Section IV lists all the necessary steps to create own parser in our methodology, Section V presents obtained results and Section VI concludes the work.

II. Related Work

Rather outdated work by Braun et al. [2] uses the onion-like structure of hand-written protocol wrappers to parse packets. However, due to the 32-bits-wide data path and an old FPGA, the parser achieves a throughput of only 2.6 Gb/s. There is no extensive concept of a common interface for module reuse, and it is unclear how the parser scales for a wider data path.

Kobierský et al. [3] describe the packet headers in XML and generate finite state machines, which parse the described protocol stack. However, the number of states in FSMs rises rapidly with the width of the data bus. Also, the crossbar used in the field extraction unit does not scale well.

While not directly related to our work, there has been an extensive research of general High-Level Synthesis systems, usually translating pure or modified imperative languages (such as C, C++, Java) into the hardware. Most of this research aims to find a potential parallelism hidden in the program loops and to make use of it by unrolling the loops and pipelining the computation. However, the *for* or *while* cycle is far from a convenient description of a packet parser, whose most natural model of computation is perhaps a directed graph of mutually dependent memory accesses. That may be the reason why we do not see many results of a general HLS in this area.

There is a general HLS result that was given by Dedek et al. in [4]. Handel-C language is used to describe the design, but the details are not disclosed. The reported speed of 1 454 Mb/s implies that a rather narrow data bus (probably 16 bit) was used. Therefore, the concern is about the scalability in terms of both an effective description in Handel-C and an effective compilation to hardware for much wider data words. This work also demonstrates that using processors for the packet parsing

978-1-4799-4558-0/14/$31.00

gives poor results. Compared to the Handel-C implementation, a custom RISC processor designed specifically for the packet parsing yields roughly the same chip area, but achieves only a half of the throughput. Using the MicroBlaze [5] processor (which is not optimized for the packet parsing) requires double resources and brings only 5.7 % throughput compared to the Handel-C solution.

A good example of a domain-specific HLS was given by Attig and Brebner in [1]. They utilize their own Packet Parsing (PP) language to describe (with the syntactic sugar of an object orientation) the structure of packet headers and the methods which define parsing rules. The description is then compiled from PP to the pipeline stages implementation. However, the results indicate that the price for a convenient design entry is the chip area and the latency – most parsers with 1024-bit datapath use over 10 % of the resource-abundant Xilinx Virtex-7 870HT FPGA [6] and the latency varies from 292 to 540 ns.

The Kangaroo system [7] uses RAM to store the packets and employs the on-chip CAM to perform a lookahead. Lookahead is the process of loading several fields from the packet memory at once, allowing to parse several packet headers in a single cycle. The dynamic programming algorithm is used to precompute data structures, so that the parsing of the longest paths in a parse tree is the most accelerated by the lookahead, as it is impractical to perform the lookahead for all the possible protocol combinations. This approach has the architectural limitation of storing the packets in the memory and accessing them afterwards. The memory soon becomes a bottleneck. Our approach, however, parses packets ”on the fly”, which means that the only packet data storage are the pipeline registers.

III. Modular Parser Design

A. Input Packet Interface

We start with the design of an input packet interface, which conveys packets into (and through) the parser. While the interface design may seem trivial, it becomes very important for high bandwidth applications. This is due to the fact that FPGAs achieve rather low frequency, roughly between 100-400 MHz. To support the bandwidth over 100 Gb/s, we must use a very wide data bus (up to 2048 bits). Since the shortest Ethernet frame is 64 Bytes (512 bits), packet aliasing and aligning become an issue. Therefore, the achievable effective bandwidth is considerably smaller than the theoretical raw bandwidth.

We propose and our packet convey protocol uses two techniques to utilize the raw bandwidth more effectively than the standard approach:

- **Partially aligned start.** The first packet byte may appear at any position aligned to eight bytes. This corresponds to the 40 and 100 Gb/s Ethernet standard. For a data bus wider than 64 bits (8 bytes), this technique allows the packet to start at other positions than the first byte of a data bus word.
- **Shared words.** One data word may contain the last bytes of the packet x and the first bytes of the packet $x + 1$. The packets may not overlap within the word and the partially aligned start condition may not be violated.

Examples of both aforementioned techniques are shown in the Fig. 1. Using these techniques we bring the effective throughput for the usual packet length distribution much closer to the theoretical limit.

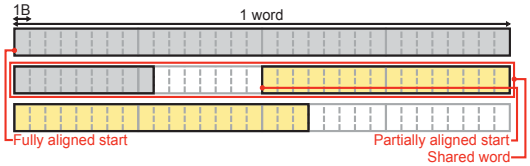

Fig. 1. The example of possible packet positions when using the proposed techniques for the better raw bandwidth utilization.

B. Parser Submodules

Since we realize that the development of VHDL modules is rather low-level and often very time-consuming, we continue with the design of *Generic Protocol Parser Interface* (GPPI). This interface provides the input information necessary to parse a single protocol header: (1) current packet data being transferred at the data bus, (2) current offset of the packet data and (3) offset of the protocol header. GPPI output information includes (4) extracted packet header field values and the information needed to parse the next protocol header: (5) offset and (6) type of the next protocol header. Fig. 2 shows how the modules are connected. By manually adhering to GPPI, we achieve a hint of object orientation in VHDL – all protocol header parsers use the same interface (except for the extracted header fields) and therefore can easily be interchanged if needed. This improves the code maintainability and enables the easy extensibility of the parser: any new protocol header parser is connected just in the same way as the others. This feature also allows an automatic connection of protocol header parsers from the high-level structure description.

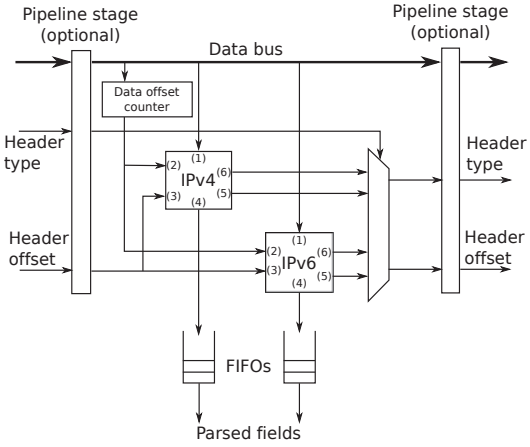

Fig. 2. Example of one pipeline stage.

The inner implementation of each protocol header parser is protocol-specific, but the basic parser block `getbyte` remains the same. This block performs waiting for a specific header field to appear at the data bus, i.e. $po + fo \in \langle do; do + dw \rangle$,

where po is the protocol header offset (module input), fo is the field offset (from the protocol specification), do is the data bus offset (module input), and dw is the data bus width. Once the header field is observed at the data bus, it is stored and can be used to compute the length of the current header, decode the type of the next header, or any other operation. Fig. 3 shows the structure of an IPv4 parser as an example.

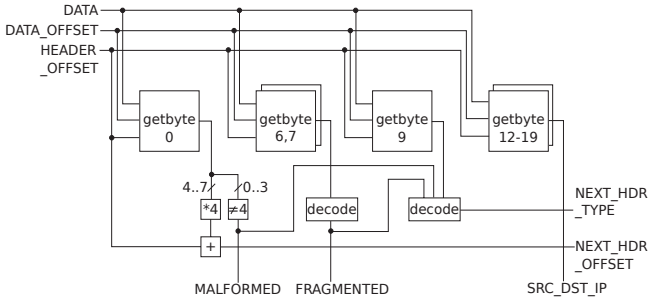

Fig. 3. Example of IPv4 protocol parser.

C. Parser Top Level

Our parser can output the information about types and offsets of protocol headers. This information is more general than just having the parsed header field values. Obtaining the header field values can be done later, externally to the parser. Our parser offers an option to skip the actual multiplexing of header field values from the data stream. This may save considerable amount of logic resources and is particularly useful for applications that read only a small number of header fields, or when packets are modified in a write-only manner.

Similarly to [1], our parser also uses pipelining to achieve high throughput. However, every pipeline step in our design is optional. If many pipelines are enabled, then the frequency (and the throughput) rises, but also the latency and used logic resources increase. By tuning the use of pipelines, designer can find the optimal parameters for the particular use case.

Each protocol parser contains an inner bypass for situations when its protocol is not present in a packet (not shown in Fig. 3). Thanks to this bypass, the protocol parser submodules can be arranged in a simple pipeline with a constant latency. This property also makes adding a support for new protocols into the parser stack very easy, without the requirement for any changes in the existing protocol parsers. Fig. 4 shows the example top level structure of the parser. Note that the inner bypasses allow to skip certain protocol headers (e.g. VLAN, MPLS), if these are not present in the packet.

The data width required for high throughput (over 100 Gb/s) may be 1024 or even more bits. This implies that there may be more packets in one data word. Our parser is able to handle such situation, provided that no two packet headers of the same type from different packets are present in one data word. For example, if the data word contains the IPv4 header (and the following bytes) of the packet A, and a part of the packet B that includes the IPv4 header, then the packet B is delayed by one cycle in our parser. This situation may only occur only for wide data buses (512 bits and more), and short packets (close to minimal length of 64 bytes) with very

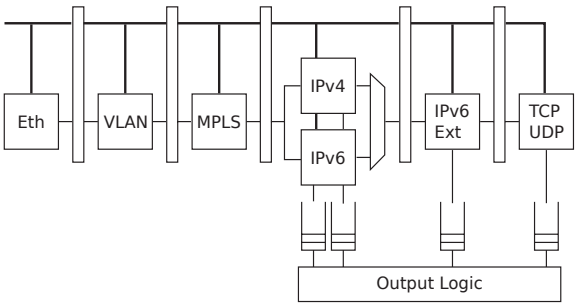

Fig. 4. Example of the parser top level structure.

short inter-packet gap. Our measurements of the real high-speed networks show that it is very rare situation.

IV. Creating the Parser According to the Application Requirements

With the description of the parser design, it is rather straightforward to create own, customized parser. We identify three basic steps:

- Parser submodules implementation
- Parser top level connection
- Parser state space search

Parser submodules implementation comprises manual writing of the VHDL code for each supported protocol. However, GPPI enables easy reuse of the submodules – once written, the protocol parser submodule can always be reused. Also, many generic building blocks of the submodules are already available, for example the `getbyte` module, which extracts a single byte at a certain offset from the packet. In general, the parser submodules for the common protocols are very similar and follow the same informal code template. For many today's protocols the parser submodule is only the `getbyte` modules with correctly configured offsets and some protocol-specific logic to compute the information about the next protocol from the extracted fields. Therefore, one can easily create a parser submodule implementation from the protocol structure specification.

There is also a space here for manual optimizations. For example, extracting a byte from the data bus using the getbyte normally requires a full multiplexer, which is able to extract a byte from any byte position in the data word (Fig. 5a). The multiplexer is controlled by the current data bus offset, the offset of a header within the packet, and the offset of the desired field within the header (which is often constant). However, given the fact that a packet may start only at certain positions in the data word, the current data offset may contain only the values with the corresponding resolution. Also, we can often derive all the possible offsets of the packet header from the analysis of all the possible orderings and sizes of the protocol headers appearing in the packet *before* the current protocol header. Combining the three values (data offset resolution, possible header offsets, constant field offset) together, we can use simpler multiplexers, which do not allow to extract fields from impossible positions. The use of simpler multiplexers in getbyte, together with the fact that getbyte

modules form the main core of the protocol analyzing and data extracting, result in significant chip area savings. For example in a classical TCP/IP protocol stack, header lengths are multiples of 4. Therefore, the size of multiplexers can be reduced 4 times and the size of the whole parsing logic by nearly the same amount. This is illustrated in the Fig. 5.

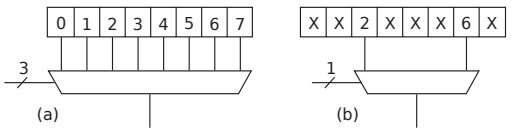

Fig. 5. Example of 64b getbyte multiplexer: full (a) and optimized (b).

Parser top level connection once again requires the designer to write VHDL. In this case, the protocol submodules are connected via GPPI pipelines to the structure corresponding to the expected order of the protocol headers in packets. Extracted header field values can be stored in output FIFOs.

Parser state space search is the final step. It takes into account other parser requirements than a set of supported protocols. The state space is created by the selective bypassing of pipelines and by setting the data width of the packet convey protocol (all easily set by generic parameters).

For example, there is often a requirement on the throughput. In that case, we are looking for a parser with throughput equal or higher than the requirement. By synthetizing a parser with all the possible settings and ruling out those which do not satisfy the throughput requirement, we obtain a set of satisfying solutions. However, the solutions will differ in the size of chip area and in latency. From this set we select a Pareto set, which contains only the dominating solutions (those for which there is no better solution in both chip area and latency). If the Pareto set has more than one member solution, we have to decide which parameter (area or latency) is more important for our application.

Generally, each candidate solution creates one point in the 3-D space with dimensions throughput, area and latency. Each pipeline step and each data width option double this space, possibly ending in a situation when the exhaustive search is no longer possible, taking into account that a single synthesis run takes time in the order of minutes. In that case we suppose that some global optimization algorithm, such as simulated annealing or a genetic algorithm can be used. Good heuristic helping these algorithms could be to rule out some of the pipeline positions, more precisely to place the pipelines evenly in the parser to create evenly long critical paths.

A. Implications for High Level Synthesis

After identifying the steps needed to be performed manually, we can now provide a list of features desirable for a good HLS, general or platform specific:

- Way to describe parser interface and protocols.
- Way to specify header formats and their dependency.
- Automatic inference of logical constraints (for multiplexer simplification etc.).
- Generator of parametrized code.
- Way to describe the design goals (area, latency etc.).
- The best fitting solution finder (exhaustive/heuristic).

Note that these requirements do not imply any particular *type* of a parser. Such HLS may generate pipelined parsers similar to ours, or the parsers based on a completely different paradigm (e.g. FSM or processor+code).

V. Results

We have implemented a parser supporting the following protocol stack: Ethernet, up to two VLAN headers, up to two MPLS headers, IPv4 or IPv6 (with up to two extension headers), TCP or UDP. (see Fig. 6). The parser is able to extract the classical quintuple: IP addresses, protocol, port numbers. Apart from that, it can also provide the information about present protocol headers and their offsets including the payload offset.

We have tested properties of the designed parser with 3 different protocol stacks:

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

For each mentioned protocol stack, one test case is done for the parser with the logic to extract the classical quintuple and one for the same parser without the extraction logic (providing only the offsets).

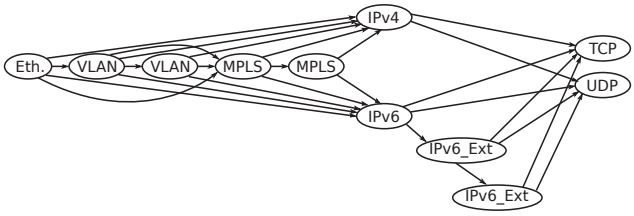

Fig. 6. Structure of supported protocols (full protocol stack).

We provide the results after a synthesis for the Xilinx Virtex-7 870HT FPGA, with different settings of the data width and the number of pipeline stages. These settings, together with the resulting frequency, latency and resource usage, generate a large space of solutions, in which the Pareto set can be found and used to pick the best-fitting solution for an application. In each test case, we use 5 different data widths: numbers from 128 to 2 048 bits that are powers of 2. For each data width, every possible placement of pipelines for the tested protocol stack is shown as a point in the graph and the Pareto set is highlighted. Points representing results for each data width are shown in different shapes and colors.

For each test case we provide 2 graphs: the first one shows the relation between throughput and FPGA resources with the Pareto set highlighted, without any regard to latency. The second graph shows the relation between throughput and latency with the Pareto set highlighted, without any regard

to FPGA resources. In the graph with the relation between throughput and FPGA resources, the second Pareto set (the lower, dashed curve) is also shown. This Pareto set shows the best achievable solutions for our parser without the quintuple extraction logic. Similar Pareto set is not shown in the graph with the relation between throughput and latency, because the usage of the quintuple extraction logic affects the latency of the parser only slightly (the critical paths are mostly in the next header computation logic).

Fig. 7 shows the throughput and the FPGA resources and Fig. 8 shows the throughput and the latency for the full protocol stack. There are 9 configurable pipeline positions in the parser implementing the full protocol stack. This leads to 512 different possible placements of pipelines in this parser for each data width. Mentioned graphs therefore show results for 2 560 different solutions with the Pareto sets highlighted.

For a comparison of the achieved Pareto set results for different protocol stacks, we provide graphs in Fig. 9 (throughput and FPGA resources) and the Fig. 10 (throughput and latency). From these figures one can clearly see that the supported protocol stack can rapidly change the parameters of the parser in terms of chip area and latency. Therefore, a careful protocol support selection is very important for the optimal result. For example, just by turning off the IPv6 support we can bring down the resource utilization by almost 50 %. Latency, on the other hand, is sensitive to the depth of the protocol stack, (see Fig. 6) therefore turning off the support for the VLAN and MPLS headers lowers the latency significantly.

A closer look at the Pareto set optimized for latency and throughput (without regard to FPGA resources) from Fig. 8 is presented in Tab. I. The last line of the table is the estimation of the parser from [1] with similar configuration of the supported protocols (TcpIP4andIP6). It is obvious that our parser can achieve much better parameters than the parser from [1].

Data Width	Pipes	Throughput [Gb/s]	Latency [ns]	LUT-FF pairs
256	0	14.5	17.1	3 238
512	0	28.4	18.0	4 053
2 048	0	96.9	21.1	17 685
2 048	1	158.5	25.9	18 547
2 048	2	212.8	28.9	18 317
2 048	4	333.0	30.8	21 775
2 048	5	352.0	34.9	22 373
2 048	7	453.0	36.2	26 728
2 048	8	478.1	38.6	29 301
1 024	?	325	309	67 902

TABLE I. PARETO SET FOR THE BEST THROUGHPUT AND LATENCY OF THE FULL PROTOCOL STACK PARSER

Next, we provide the data for the example from the Section IV: Given a set of supported protocols and the target throughput, find all solutions in the Pareto set. We use three sets of supported protocols mentioned earlier and the target throughputs of 40, 100 and 400 Gb/s. All nine Pareto sets are shown in the Fig. 11. Note that while there are several solutions with the throughput over 400 Gb/s, there is only one 400 Gb/s Pareto solution for each protocol set, which means that the other solutions are not better in terms of FPGA resources nor latency. For the other target throughputs, the designer can choose the appropriate solution according to application priorities.

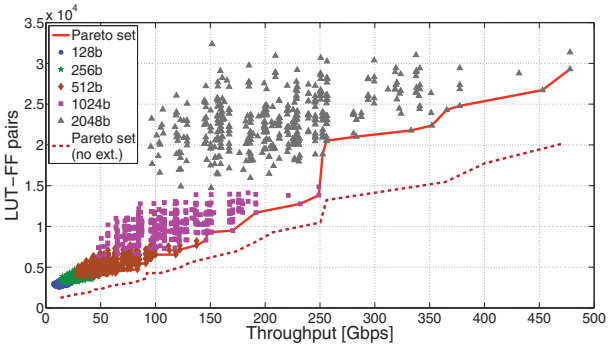

Fig. 7. The FPGA resource utilization for different settings of the full parser.

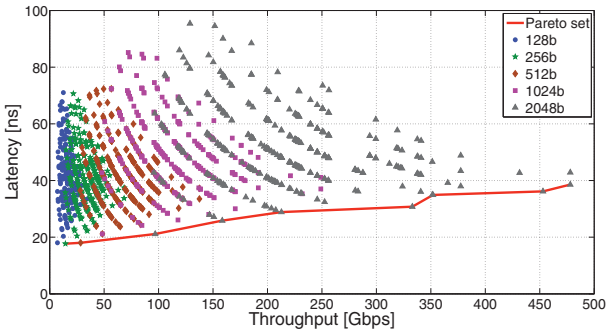

Fig. 8. The latency for different settings of the full parser.

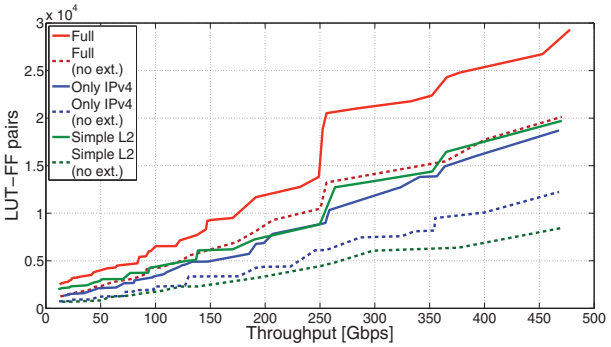

Fig. 9. Comparison of the FPGA resource utilization versus throughput Pareto sets for the tested protocol stacks.

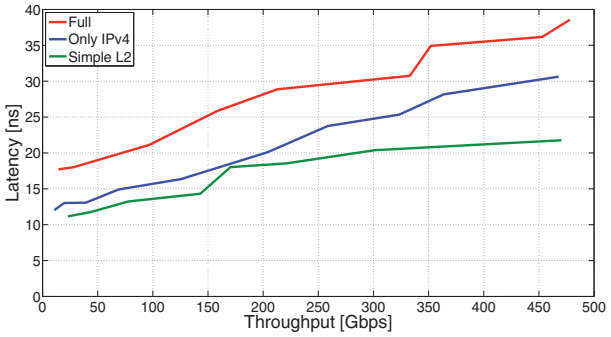

Fig. 10. Comparison of the latency versus throughput Pareto sets for the tested protocol stacks.

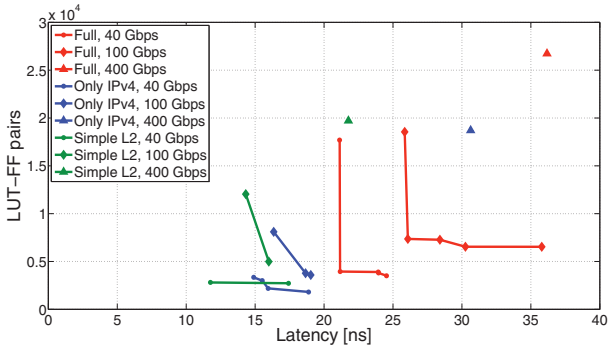

Fig. 11. Pareto sets for three given protocol sets and three target throughputs.

A careful design space exploration is very important for our parser. For example, the parser of the full protocol stack optimized for the latency uses 17 685 LUT-FlipFlop pairs to achieve near 100 Gb/s throughput with the latency of only 21.1 ns (see Tab. I), while the parser optimized for resources uses only 6 536 LUT-FlipFlop pairs to achieve the throughput just over 100 Gb/s, but with the latency of 35.8 ns (see Fig. 11).

Finally, Fig. 12 illustrates the complete Pareto set of solutions in the (latency, throughput, area) space for the full protocol stack. To create the 3D surface in the figure, the bottom (latency, throughput) plane was divided into rectangles of sizes (1 ns $\times$ 10 Gb/s) and the smallest solution that satisfies the required latency and throughput was found for each rectangle. Therefore, each horizontal level of the surface represents one solution from the Pareto set. Finer-grained division of the (latency, throughput) plane would result in more solutions, but also in less readable image.

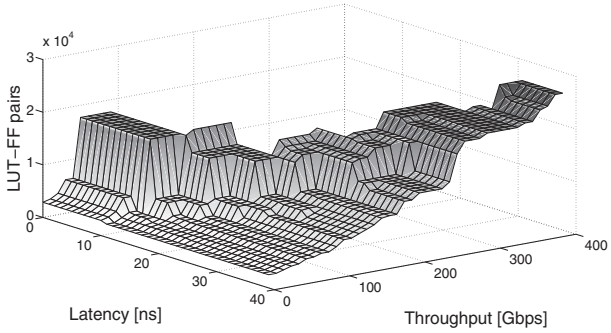

Fig. 12. 3D surface plot of the Pareto set

VI. Conclusion

This paper introduces a highly configurable packet parser for FPGA, which achieves throughput in the range of tens to hundreds of Gb/s and is usable in a variety of applications. The key concept is a selective pipelining, which allows to find the best fitting solution with regards to the requirements. The parser uses only 1.19 % of the Virtex-7 870HT FPGA resources to achieve a throughput over 100 Gb/s and 4.88 % for a throughput over 400 Gb/s, which leaves most of the FPGA resources free for implementing other functions of target applications.

This work also presents the methodology of a modular parser design and demonstrates the need for a thorough exploration of the solution space. Moreover, it suggests several capabilities that a High-Level Synthesis system should possess to succeed in area of packet parsers creation.

Acknowledgement

This research has been supported by the “CESNET Large Infrastructure” project no. LM2010005 funded by the Ministry of Education, Youth and Sports of the Czech Republic, the “DMON100” project no. TA03010561 funded by the Technology Agency of the Czech Republic, BUT project FIT-S-14-2297 and the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070.

References

[1] M. Attig and G. Brebner, “400 gb/s programmable packet parsing on a single fpga,” in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, oct. 2011, pp. 12–23.

[2] F. Braun, J. Lockwood, and M. Waldvogel, “Protocol wrappers for layered network packet processing in reconfigurable hardware,” *Micro, IEEE*, vol. 22, no. 1, pp. 66–74, 2002.

[3] P. Kobierský, J. Kořenek, and L. Polčák, “Packet header analysis and field extraction for multigigabit networks,” in *Proceedings of the 2009 12th International Symposium on Design and Diagnostics of Electronic Circuits&Systems*, ser. DDECS. Washington, USA: IEEE Computer Society, 2009, pp. 96–101.

[4] T. Dedek, T. Martínek, and T. Marek, “High level abstraction language as an alternative to embedded processors for internet packet processing in fpga,” in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, aug. 2007, pp. 648–651.

[5] “Xilinx microblaze soft processor,” Xilinx, Inc., http://www.xilinx.com/tools/microblaze.htm.

[6] “Xilinx virtex–7 fpga family,” Xilinx, Inc., http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.

[7] C. Kozanitis, J. Huber, S. Singh, and G. Varghese, “Leaping multiple headers in a single bound: Wire-speed parsing using the kangaroo system,” in *IEEE INFOCOM*, mar. 2010.

A.3 Paper III

Memory Efficient IP Lookup in 100 Gbps Networks

MEMORY EFFICIENT IP LOOKUP IN 100 GBPS NETWORKS

Jiří Matoušek

CESNET, z. s. p. o.
Zikova 4,
Praha 6, 160 00,
Czech Republic
imatousek@fit.vutbr.cz

Martin Skačan, Jan Kořenek

IT4Innovations Centre of Excellence
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, Brno, 612 66, Czech Republic
xskaca00@stud.fit.vutbr.cz, korenek@fit.vutbr.cz

ABSTRACT

The increasing number of devices connected to the Internet together with video on demand have a direct impact to the speed of network links and performance of core routers. To achieve 100 Gbps throughput, core routers have to implement IP lookup in dedicated hardware and represent a forwarding table using a data structure, which fits into the on-chip memory. Current IP lookup algorithms have high memory demands when representing IPv6 prefix sets or introduce very high pre-processing overhead. Therefore, we performed analysis of IPv4 and IPv6 prefixes in forwarding tables and propose a novel memory representation of IP prefix sets, which has very low memory demands. The proposed representation has better memory utilization in comparison to the highly optimized Shape Shifting Trie (SST) algorithm and it is also suitable for IP lookup in 100 Gbps networks, which is shown on a new pipelined hardware architecture with 170 Gbps throughput.

1. INTRODUCTION

The increasing speed of network links has a direct impact to the design and performance of core routers. To achieve 100 Gbps throughput, core routers need dedicated hardware for IP lookup in a forwarding table. Moreover, the size of forwarding tables is increasing with the amount of devices and networks connected to the Internet [1]. It means that core routers have to perform faster IP look up in larger forwarding tables.

The most demanding part of IP packet forwarding is the Longest Prefix Match (LPM) operation. It implements lookup of the longest prefix from a forwarding table, which corresponds to the destination IP address of a packet. For example, let us consider the prefix set from Fig. 1 and a packet with the 8-bit destination address $IP = 11100010$. In this case, prefixes $P1$, $P4$, and $P7$ correspond to the destination address. However, since the prefix $P7$ is the longest one among these prefixes, it is the only result of the LPM operation.

Core routers supporting 100 Gbps throughput have to be able to perform more than 150 million lookups per second (MLPS). Therefore, a new LPM result has to be provided every 6.72 ns. It is possible to achieve such lookup performance only with hardware implementation of the LPM operation [2]. However, in such a case there is usually a bottleneck in relatively slow access to the external memory, where a prefix set extracted from a forwarding table is stored. This can be solved by storing the prefix set in the easily accessible on-chip memory. Nevertheless, the on-chip memory has a limited capacity, therefore the prefix set has to be represented using a memory efficient data structure.

In this paper we propose a novel memory efficient representation of prefix sets, which can be stored in the on-chip memory with a limited capacity. The proposed representation was designed according to analysis of different prefix sets (real IPv4 and IPv6, generated IPv6). We also propose a pipelined hardware architecture, which utilize the designed prefix set representation and is able to perform more than 150 MLPS.

The rest of the paper is organized as follows. Section 2 contains a brief summary of related LPM algorithms. Section 3 describes performed analysis and its results. The proposed novel prefix set representation is introduced in section 4 and the hardware architecture for its processing is described in section 5. Next section 6 shows results of the performed experiments. Conclusion of the paper and remarks about our future work are in section 7.

2. RELATED WORK

The LPM operation is in many commercial devices implemented using (TCAM) *Ternary Content-Addressable Memory*. Such implementation is able to provide an LPM result in just one clock cycle, but TCAMs are expensive, power-

This work was supported by the grant TAČR TA03010561, the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070, the research program MSM 0021630528, and the grant BUT FIT-S-11-1.

978-1-4799-0004-6/13/$31.00 ©2013 IEEE

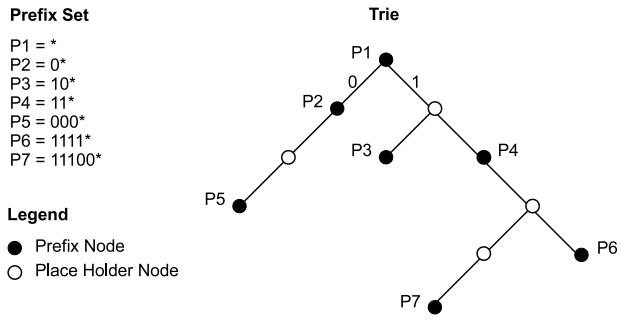

Fig. 1. Sample Prefix Set Represented by Trie

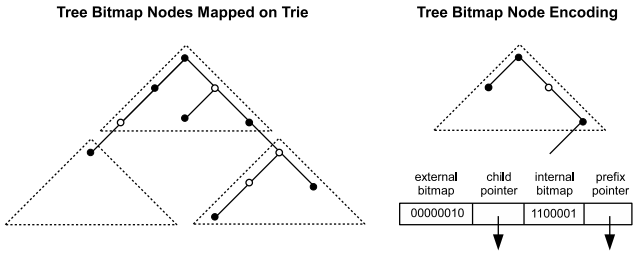

Fig. 2. Tree Bitmap Mapping and Encoding ($SL = 3$)

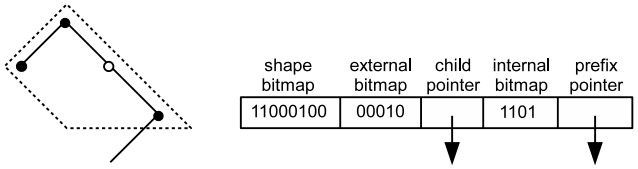

Fig. 3. Shape Shifting Trie Node Encoding ($K = 4$)

hungry and slow in updating their content. Therefore, many algorithmic solutions to LPM have been proposed [3], [4], [5], [6].

A basic data structure utilized in the majority of LPM algorithms is called *trie* [3]. It is a binary tree with prefixes encoded into its structure. The root node of a trie represents the empty prefix. Left and right child nodes of any trie node represent prefixes created from their parent's prefix by appending 0 and 1, respectively. Trie nodes representing prefixes from a prefix set are called prefix nodes, while other trie nodes are referred to as place holder nodes. The representation of the sample prefix set using the trie is shown in Fig. 1. The LPM operation using a trie data structure is performed by traversing a trie from the root to leaves according to bit values of packet's destination address taken from the most significant bit to the least significant bit. The last prefix node visited during such a traversal represents the longest matching prefix.

Adding and removing prefixes from a trie can be done using standard operations on a binary tree. Performing the LPM operation on a trie is also straightforward. However, only one bit of the input can be processed in each step, which means the worst case performance of 32 and 64 steps for IPv4 and IPv6 prefixes, respectively. A trie data structure also has high memory demands, which are caused mainly by the high number of pointers in a trie.

In order to increase lookup performance of trie-based LPM algorithms, multibit tries have been designed. One of the best known multibit trie algorithm is called the *Tree Bitmap* (TBM) [4]. This algorithm represents a set of prefixes using a 2^{SL}-tree, where the parameter SL (i. e. stride length) determines the number of input bits processed in each step of TBM. Mapping of TBM nodes with $SL = 3$ on the trie from Fig. 1 is shown on the left hand side of Fig. 2. On the right hand side of the same figure, there is a sample TBM node and its encoding using two bitmaps and two pointers. The external bitmap contains 2^{SL} bits and it determines the presence of child nodes, while the internal bitmap with $2^{SL} - 1$ bits contains information about prefixes represented by the TBM node. Child and prefix pointers refer to information about child nodes and prefix-related data, respectively.

Use of bitmaps for encoding of a TBM node makes this algorithm easy to implement in hardware. Moreover, the compact representation of a TBM node allows it to be read from a memory in just one clock cycle. The fixed structure of a node is advantageous when updates of a prefix set are performed, however, it may cause high memory overhead in a sparse prefix tree.

Another multibit trie algorithm is called the *Shape Shifting Trie* (SST) [5]. This algorithm is based on TBM, but it tries to overcome its main drawback by introducing adaptive shape of a node, which reduces memory overhead in sparse prefix trees. Adaptive shape is allowed by the shape bitmap consisting of $2K$ bits (see Fig. 3). The parameter K determines the maximum number of underlying trie nodes, that can be represented by a single SST node. SST has exceptionally low memory demands, but its computational complexity is usually unacceptable. Moreover, to the best of our knowledge, there is no hardware architecture implementing SST.

A representation of prefix set with low memory demands and a hardware architecture for its processing, which provides lookup performance higher than 150 MLSP, has been introduced in [6]. We will further refer to this algorithm as the *Prefix Partitioning Lookup Algorithm* (PPLA). PPLA uses a trie data structure, but a trie is utilized only for partitioning a set of prefixes into several disjoint subsets. Each subset is then represented using a separate binary search tree or a 2-3 tree data structure and processed in a separate processing pipeline. This algorithm has good memory efficiency – it needs approximately only one byte of memory to store one byte of IPv4 or IPv6 prefix. However, the proposed representation causes linear growth of mem-

Table 1. Details of Used Prefix Sets

Prefix Set	Prefixes	Source	Date
IPv4			
rrc00	332 118	http://data.ris.ripe.net/	2010-06-03
IPv4-space	220 779	http://bgp.potaroo.net/	2011-12-21
route-views	442 748	http://archive.routeviews.org/	2012-09-20
IPv6			
AS1221	10 518	http://bgp.potaroo.net/	2012-09-21
AS6447	10 814	http://bgp.potaroo.net/	2012-09-21
Generated IPv6			
rrc00_ipv6	319 998	generated using [7] from rrc00	
IPv4-space_ipv6	150 157	generated using [7] from IPv4-space	
route-views_ipv6	439 880	generated using [7] from route-views	

Table 2. Memory Demands of Different LPM Algorithms

Prefix Set	Prefixes	Memory Demands [Kb]		
IPv4		**Trie**	**TBM (*SL=5*)**	**SST (*K=32*)**
rrc00	332 118	47 639.7	9 689.4	6 930.4
IPv4-space	220 779	24 252.4	5 702.1	4 081.0
route-views	442 748	62 650.5	11 942.1	8 775.0
IPv6		**Trie**	**TBM (*SL=3*)**	**SST (*K=32*)**
AS1221	10 518	3 518.3	1 076.9	588.5
AS6447	10 814	3 673.8	1 125.1	617.1
Generated IPv6		**Trie**	**TBM (*SL=4*)**	**SST**
rrc00_ipv6	319 998	307 641.5	87 257.1	N/A
IPv4-space_ipv6	150 157	153 877.3	43 958.7	N/A
route-views_ipv6	439 880	418 663.7	118 889.4	N/A

ory demands with the number of represented prefixes and initial partitioning of a prefix set introduces very high preprocessing overhead.

Linear dependence of memory demands on the number of represented prefixes is one of the most significant issues connected with PPLA. In trie-based LPM algorithms, nodes close to the root of a tree are shared by several prefixes. This property should allow to represent one byte of prefix using less than one byte of memory. Therefore, we focus our analysis on previously introduced trie-based algorithms.

3. ANALYSIS

Basic information about prefix sets extracted from forwarding tables of core routers, which we use in our analysis, are summarized in Table 1. In order to obtain results relevant for many different situations, we use sets of real IPv4 and IPv6 prefixes as well as sets of IPv6 prefixes generated using [7]. Moreover, diversity of data for analysis is increased by using real IPv4 and IPv6 sets from different sources and acquired on different days. Experiments with prefix sets were performed using Netbench tool [8].

The first part of analysis was focused on memory demands of Trie, TBM, and SST algorithms and its results are shown in Table 2. Parameters SL and K of TBM and SST algorithms, respectively, were chosen with respect to the minimum memory demands. As can be seen, K was set to the same value for all prefix sets, while SL was set to a different value for each group of prefix sets. This reflects different density of the prefix tree (smaller value of SL means lower density) between groups of prefix set. Missing results of SST memory demands for generated IPv6 prefix sets cannot be provided because of very high computational complexity of SST.

Table 2 shows, that the lowest memory demands can be achieved when SST is used, while the highest memory demands are connected with the Trie algorithm. Such results would propose SST to be a candidate for further optimization of memory demands. However, as stated in section 2, SST suffers from high computational complexity and there is no hardware architecture for this algorithm. Therefore, the next part of our analysis was focused on identification of possibilities for optimization of TBM's memory demands.

Table 3. Classification of Nodes From a TBM Representation of route-views (434 552 Nodes, $SL = 3$)

Prefixes	Child Nodes								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	**26 829**	**11 859**	**6 876**	**5 422**	**3 679**	**3 547**	**4 297**	**14 138**
1	**278 804**	6 220	4 244	2 840	4 463	1 683	2 416	876	2 051
2	**21 005**	3 270	4 198	1 599	2 688	724	792	393	842
3	**5 716**	1 093	2 000	596	806	293	286	160	306
4	**3 786**	447	543	220	322	106	129	102	267
5	**679**	63	55	22	48	20	25	25	78
6	**298**	30	22	9	23	3	9	6	64
7	**70**	6	3	3	8	4	3	7	46

Table 4. Classification of Nodes From a TBM Representation of AS1221 (25 063 Nodes, $SL = 3$)

Prefixes	Child Nodes								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	**11 303**	**1 666**	**812**	**538**	**184**	**145**	**131**	**249**
1	**8 965**	547	142	19	17	3	2	1	1
2	**193**	21	14	4	3	0	1	0	0
3	**50**	3	3	3	1	0	1	0	0
4	**29**	3	1	1	3	1	1	0	0
5	**0**	1	0	1	0	0	0	0	0

TBM analysis was performed by classification of TBM nodes according to the number of child nodes and the number of prefixes represented by a TBM node. Results of this classification for selected IPv4, IPv6 and generated IPv6 prefix sets are shown in Tables 3, 4, and 5, respectively. Even though all tables show classification of TBM nodes with $SL = 3$, presented results can be used for identification of general trends in TBM.

Analysis of the TBM representation of all selected prefix sets shows two significant groups of node. The first group contains leaf nodes (the leftmost column in Tables 3, 4, and 5), while the second group contains internal nodes without prefixes (the first row in Tables 3, 4, and 5). Therefore, efficient encoding of nodes from these two groups will significantly reduce memory demands of TBM.

Table 5. Classification of Nodes From a TBM Representation of route-views_ipv6 (2 239 971 Nodes, $SL = 3$)

Prefixes	Child Nodes								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	**1 597 683**	**143 258**	**39 958**	**21 056**	**9 332**	**5 637**	**3 958**	**4 462**
1	**406 100**	3 503	746	263	108	45	15	10	6
2	**2 623**	171	118	40	39	21	8	6	1
3	**475**	37	24	5	7	3	1	3	1
4	**155**	11	7	3	1	1	0	0	0
5	**44**	1	4	3	2	1	0	0	0
6	**12**	0	1	0	0	0	0	0	0
7	**2**	0	0	0	0	0	0	0	0

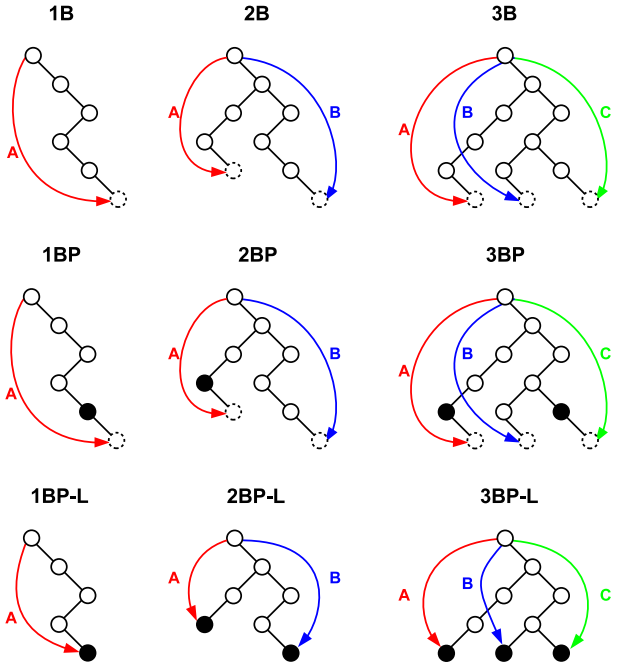

Fig. 4. Newly Proposed Types of Node

4. PREFIX SET REPRESENTATION

Performed analysis has shown two groups of TBM nodes, whose more efficient encoding could reduce memory demands of TBM. To this end, we propose a representation of prefix set using thirteen different types of node. These thirteen types can be divided into two groups – nine newly proposed nodes (see Fig. 4) and four variants of a TBM node. Properties of nodes in Fig. 4 are reflected in their name. A node can encode 1 branch (1B), 2 branches (2B) or 3 branches (3B) of an underlying trie. It can also contain a prefix node (P), but such a prefix node is allowed only in the lowest level of the node and it may not occur in all branches. Presence of a prefix node in the lowest level of all branches is compulsory only in the case of leaf nodes (L). Used TBM nodes include a standard node for $SL = 3$ (TBM3) and leaf TBM nodes for $SL = 3, 4, 5$ (TBM3-L, TBM4-L, TBM5-L).

Table 6. Basic Parameters of Different Types of Node When Aligned to 8-bit and 16-bit Boundary

Node Type	Size Aligned to 8 bits			Size Aligned to 16 bits		
	Branch Length [bits]	**Unaligned Size** [bits]	**Aligned Size** [bits]	**Branch Length** [bits]	**Unaligned Size** [bits]	**Aligned Size** [bits]
1B	24	56	56	17	48	48
1BP	19	72	72	13	64	64
1BP-L	20	48	48	20	48	48
2B	16	72	72	14	64	64
2BP	10	80	80	11	80	80
2BP-L	12	55	56	15	61	64
3B	11	78	80	12	78	80
3BP	5	80	80	6	80	80
3BP-L	7	53	56	9	62	64
TBM3	3	75	80	3	67	80
TBM3-L	3	30	32	3	30	48
TBM4-L	4	38	40	4	38	48
TBM5-L	5	54	56	5	54	64

In order to make hardware implementation of the proposed representation feasible, it is necessary to align the size of node representations to some boundary. A smaller boundary implies smaller memory overhead but higher number of different sizes of node, hence higher utilization of resources for processing such data structures. Therefore, we consider two different alignments (to the 8-bit and 16-bit boundary), which should allow us to achieve a reasonable compromise between memory overhead and resources utilization. We will examine real memory demands and resources utilization for both alignments.

Basic parameters of different types of node, when aligned to the 8-bit and 16-bit boundary, are summarized in Table 6. The size of a node is determined mainly by the maximum branch length and the presence of child and prefix pointers. The maximum branch length, which is the same for all branches in a node, is shown in Table 6. The prefix pointer encoded on 19 bits is present only in nodes, that can represent prefixes, i. e. nodes with P in their name and TBM nodes. The child pointer is encoded on 23 bits (in the case of alignment to the 8-bit boundary) or 22 bits (in the case of the 16-bit boundary) and it is present in all non-leaf nodes, i. e. nodes without L in their name. Since Table 6 shows aligned as well as unaligned size of each type of node, memory overhead introduced by alignment can be computed as difference of these two values.

The mapping of proposed nodes on a trie is done according to the algorithm in Fig. 5. This algorithm uses, except standard queue operations ENQUEUE and DEQUEUE, three auxiliary functions. MAP_COST returns the cost of mapping of given type of node from the specified position in the trie. The cost is determined using equation (1), where p is the number of covered prefix nodes, n is the number of all covered trie nodes, and $size$ is the size of given type of node. Mapping of the selected type of node to the specified posi-

Input: pointer $root$ pointing to the root node of the trie
Output: pointer $root$ pointing to the root node of the mapped tree

```
1:  Q ← ∅
2:  if root ≠ NULL then
3:      ENQUEUE(Q, root)
4:  while Q ≠ ∅ do
5:      trie ← DEQUEUE(Q)
6:      max_cost ← 0
7:      best_type ← NULL
8:      for each type ∈ node_types do
9:          cost ← MAP_COST(type, trie)
10:         if cost > max_cost then
11:             max_cost ← cost
12:             best_type ← type
13:     trie ← MAP(best_type, trie)
14:     for each child ∈ CHILDREN(trie) do
15:         ENQUEUE(Q, child)
```

Fig. 5. Pseudocode of the Mapping Algorithm

tion in the trie is done using MAP function and CHILDREN returns a list of child nodes of the given node.

$$cost = \begin{cases} \frac{p}{size} & \text{if } \frac{p}{size} > 0 \\ \frac{n}{size} & \text{otherwise} \end{cases} \quad (1)$$

5. HARDWARE ARCHITECTURE

Since the proposed representation of prefix set can be classified as multibit trie approach to LPM, a matching result is in the worst case available after processing of n nodes, where n is the height of the tree, which represents the prefix set. In order to achieve lookup performance of 150 MLSP, it is necessary to employ a processing pipeline, where each processing element (PE) performs one step of the LPM algorithm.

We propose the hardware architecture in Fig. 6 for processing of our representation of prefix set. This architecture consists of two processing pipelines with uniform PEs and dual port memory blocks shared between PEs from corresponding stages. By utilization of the dual port memory, we can achieve double performance of a single pipeline architecture without compromising on memory access. The memory block for each pipeline stage contains two parallel memories, each of which has data width of 80 bits (the maximum size of a node, see Table 6). Use of two parallel memories allows to read the whole node in one clock cycle, even if it is stored in two consecutive data words.

A high-level architecture of one PE is also shown in

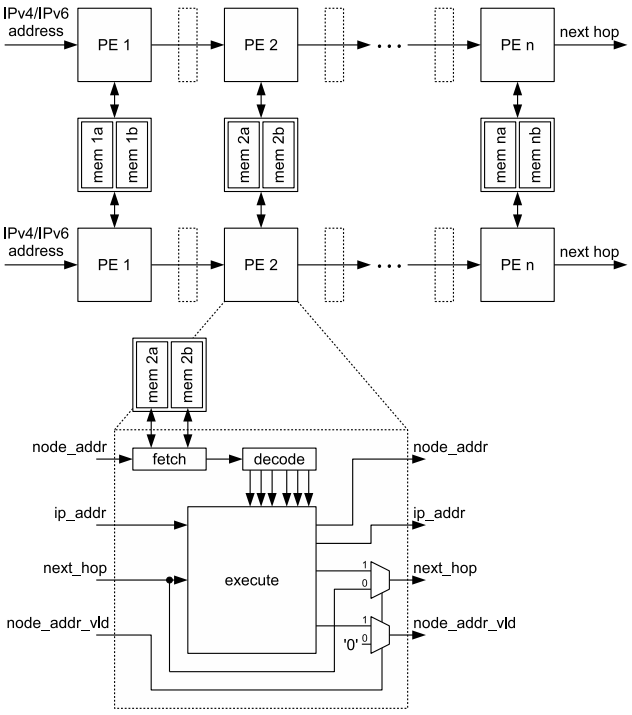

Fig. 6. Double Processing Pipelines With Detail of One Processing Element (PE)

Fig. 6. Processing of a node in PE is like a processing of an instruction in a standard CPU. First of all, PE *fetches* the node from the memory. The representation of the node is then *decoded* and sent in parallel to the *execute* submodule. The internal structure of the execute part is shown in Fig. 7. The main part of execution is done in branch A proc, branch B proc, branch C proc, and TBM node proc submodules. The first three of them are dedicated for processing of corresponding branches of newly proposed types of node (branches are marked by letters A, B, and C in Fig. 4), while the TBM node proc submodule is dedicated for processing of TBM nodes. Since processing of different branches and different types of node is done in parallel, the select branch and the select result modules are used to select correct values for outputs of PE.

Combinatorial logic of fetch and execute submodules of PE is relatively complex. Therefore, in order to achieve desired lookup performance, it is necessary to use intra-stage registers within these two submodules. Each of them contains two sets of internal registers. In total, each PE contains four sets of intra-stage registers, thus processing a node within one PE is done in five clock cycles.

Section 4 describes two variants of node alignment in a memory – to the 8-bit or 16-bit boundary. Both variants can be processed using conceptually the same hardware architecture with only some minor changes in fetch, decode, and execute submodules. Different node alignment has the

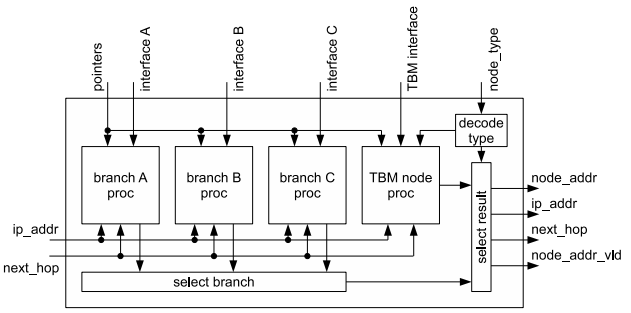

Fig. 7. Internal Structure of PE's Execute Part

biggest influence on data reorder logic in the fetch module. It also has to be reflected in the decode submodule by different interconnection of decoding logic. Relatively the smallest changes have to be done in the execute module, where it is sufficient to change data width of some internal buses.

6. EXPERIMENTAL RESULTS

First of all, we have measured memory demands of the proposed representation of prefix set on our sample IPv4 and IPv6 (both real and generated) prefix sets. The measurement has been done for both variants of node alignment in a memory and its results are presented in Table 7. Results show lower memory demands in the case of 8-bit node alignment. The difference between memory demands of the representation with nodes aligned to 8 bits and 16 bits is the most evident for IPv4 prefix sets. This is because of different density of prefix trees in their leaf part. The prefix tree of IPv4 sets is denser than the tree in the case of IPv6 sets. Therefore, leafs of the IPv4 tree are represented mainly by TBM nodes (which introduce the highest memory overhead when aligned to the 16-bit boundary), while leafs of the IPv6 tree are represented mainly by newly proposed nodes (which introduce almost the same memory overhead for both 8-bit and 16-bit alignment). Table 7 also shows the height of the tree, which represents particular prefix sets.

Both variants of the proposed architecture have been implemented for a Xilinx Virtex-6 XC6VSX475T FPGA. Utilization of resources and maximum frequency after place & route using Xilinx ISE 14.3 are shown in Table 8. As can be seen, the main difference between two proposed architectures is in the number of utilized LUTs, where the 16-bit architecture shows better results. This is mainly due to smaller data width of some buses in this architecture. The number of utilized registers is practically the same and maximum frequency is little higher in the case of the 8-bit architecture. Table 8 contains information about utilization of resources by one PE, the complete processing pipeline and also by the whole proposed architecture, which consists of two processing pipelines. Even though the length of each pipeline

Table 7. Memory Demands and Tree Height of the Proposed Representation on Different Prefix Sets

		Memory Demands [Kb]		
Prefix Set	**Prefixes**	**8-bit Alignment**	**16-bit Alignment**	**Tree Height**
IPv4				
rrc00	332 118	6 330.8	7 287.6	12
IPv4-space	220 779	3 571.4	4 297.4	12
route-views	442 748	7 779.8	9 039.6	12
IPv6				
AS1221	10 518	475.8	489.0	18
AS6447	10 814	493.8	506.6	23
Generated IPv6				
rrc00_ipv6	319 998	21 264.3	21 373.2	21
IPv4-space_ipv6	150 157	10 412.2	10 421.4	18
route-views_ipv6	439 880	29 039.5	29 207.4	20

Table 8. Resources Utilization and Maximum Frequency of Proposed Hardware Architecture (Xilinx ISE 14.3, Virtex-6 XC6VSX475T)

8-bit Alignment	**LUTs (% of All)**	**Registers (% of All)**	**Frequency [MHz]**
1 PE	3 647 (1.23 %)	1 825 (0.31 %)	127.162
1 pipeline (23 PEs)	83 881 (28.19 %)	41 957 (7.05 %)	127.162
2 pipelines (46 PEs)	167 762 (56.37 %)	83 950 (14.11 %)	127.162
16-bit Alignment	**LUTs (% of All)**	**Registers (% of All)**	**Frequency [MHz]**
1 PE	3 194 (1.07 %)	1 817 (0.31 %)	123.183
1 pipeline (23 PEs)	73 462 (24.69 %)	41 791 (7.02 %)	123.183
2 pipelines (46 PEs)	146 924 (49.37 %)	83 582 (14.04 %)	123.183

(23 PEs) allows processing the prefix set represented by the highest tree (real IPv6 set AS6447, see Table 7), the whole architecture fits into the target FPGA.

Since resources utilized by both variants of the hardware architecture are significantly lower than resources available in the target FPGA, selection of "better" variant is governed mainly by their memory demands, whose optimization is the main objective of this work. Therefore, we select the representation of prefix set with nodes aligned to the 8-bit boundary.

The selected variant can also operate on a little higher frequency, which implies higher lookup performance. Both processing pipelines are able to provide one matching result in each clock cycle, which translates into total lookup performance of almost 255 MLPS. Thus, the proposed solution is able to support throughput of 170 Gbps. Frequency of the proposed solution also determines, together with the number of pipeline stages, the overall latency. As stated in section 5, each PE consists of five pipeline stages. Therefore, the whole pipeline contains $5 \times 23 = 115$ stages. Since pro-

Table 9. Memory Demands of the Proposed Representation of Prefix Set and its Comparison to TBM and SST

Prefix Set	Prefixes	Memory [Kb]	Savings	
IPv4		**New Nodes**	**TBM (*SL=5*)**	**SST (*K=32*)**
rrc00	332 118	6 330.8	34.67 %	8.65 %
IPv4-space	220 779	3 571.4	37.37 %	12.49 %
route-views	442 748	7 779.8	34.85 %	11.34 %
IPv6		**New Nodes**	**TBM (*SL=3*)**	**SST (*K=32*)**
AS1221	10 518	475.8	55.82 %	19.16 %
AS6447	10 814	493.8	56.11 %	19.98 %
Generated IPv6		**New Nodes**	**TBM (*SL=4*)**	**SST**
rrc00_ipv6	319 998	21 264.3	75.63 %	N/A
IPv4-space_ipv6	150 157	10 412.2	76.31 %	N/A
route-views_ipv6	439 880	29 039.5	75.57 %	N/A

Table 10. Memory Efficiency Ratio (Bytes of Memory/Bytes of Prefixes) of the Proposed Representation of Prefix Set, TBM and SST

Prefix Set	Prefixes			
IPv4		**New Nodes**	**TBM (*SL=5*)**	**SST**
rrc00	332 118	0.610	0.934	0.668
IPv4-space	220 779	0.518	0.826	0.592
route-views	442 748	0.562	0.863	0.634
IPv6		**New Nodes**	**TBM (*SL=3*)**	**SST (*K=32*)**
AS1221	10 518	0.724	1.638	0.895
AS6447	10 814	0.731	1.665	0.913
Generated IPv6		**New Nodes**	**TBM (*SL=4*)**	**SST (*K=32*)**
rrc00_ipv6	319 998	1.063	4.363	N/A
IPv4-space_ipv6	150 157	1.109	4.684	N/A
route-views_ipv6	439 880	1.056	4.324	N/A

cessing in one stage takes 7.86 ns, the overall latency of the proposed solution is 903.90 ns. The overall latency also determines the size of the buffer for packets waiting for the LPM result, which has to be at least 8.6 KB for 100 Gbps Ethernet link.

Comparison of our prefix set representation, TBM, and SST in terms of memory demands is provided in table 9. Except memory demands of our solution, we show its savings compared to other LPM algorithms. The proposed prefix set representation overcomes both TBM and SST, but reduction of memory demands is higher for TBM (between 34.67 % and 76.31 %) than for SST (between 8.65 % and 19.98 %). Moreover, it is shown that the sparse prefix tree of IPv6 prefix set allows higher savings, which is due to higher utilization of memory efficient newly proposed types of node (see Fig. 4).

In order to compare memory efficiency of the proposed prefix set representation with PPLA, we provide a memory efficiency ratio (bytes of memory required to store one byte of prefix) of our solution on different prefix sets in Table 10. The value of this parameter is shown also for TBM and SST. According to [6], the average memory efficiency ratio of PPLA on generated IPv6 prefix sets is 1.01 when a 2-3 tree data structure is used. Therefore, our solution is comparable to PPLA on generated IPv6 prefix sets. However, our prefix set representation is significantly better than PPLA on IPv4 prefix sets, where [6] reports the average memory efficiency ratio of 1.00. Moreover, both TBM and SST, which were not taken into account in [6], shows better memory efficiency than PPLA on IPv4 sets. Memory efficiency of our solution and PPLA on real IPv6 prefix sets cannot be compared, because this value is not reported in [6].

Since both our solution and TBM are based on the trie, they should achieve better (i.e. lower) memory efficiency ratio on prefix sets with high number of prefixes (generated IPv6), than on prefix sets with a small number of prefixes (real IPv6). However, according to the results presented in Table 10, this is not true in our case. The most probable explanation of this situation is that IPv6 prefix sets generator [7] does not model the process of assigning IPv6 addresses correctly. We have used this generator in order to be able to compare our results with results presented in [6].

7. CONCLUSION AND FUTURE WORK

The paper proposed a novel representation of IP prefix sets using thirteen different types of node designed for a memory efficient representation of the most common situations in a prefix tree. This prefix set representation has significantly lower memory demands than TBM and it also overcomes the SST algorithm. Moreover, the proposed representation shows better memory efficiency than PPLA on real IPv4 prefix sets and comparable results on generated IPv6 prefix sets. Memory efficiency of the proposed representation and PPLA on real IPv6 prefix sets cannot be compared.

We also introduced a pipelined hardware architecture, which utilizes the proposed prefix set representation. The architecture was implemented on Xilinx Virtex-6 FPGA with 170 Gbps throughput.

As future work, we want to optimize resources utilization and lookup performance of the proposed architecture. We would also like to utilize dynamic partial reconfiguration for allocation of memory blocks to particular pipeline stages according to the actual prefix set.

8. REFERENCES

[1] (2013, Jan.) IPv6 / IPv4 Comparative Statistics. [Online]. Available: http://bgp.potaroo.net/v6/v6rpt.html

[2] M. Á. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8–23, Mar. 2001, ISSN 0890-8044.

[3] E. Fredkin, "Trie Memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, Sept. 1960, ISSN 0001-0782.

[4] W. Eatherton, G. Varghese, and Z. Dittia, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, Apr. 2004, ISSN 0146-4833.

[5] H. Song, J. Turner, and J. Lockwood, "Shape Shifting Tries for Faster IP Route Lookup," in *Proc. of the 13th IEEE International Conference on Network Protocols (ICNP'05)*. IEEE Computer Society, 2005, pp. 358–367, ISBN 0-7695-2437-0.

[6] H. Le and V. K. Prasanna, "Scalable Tree-based Architectures for IPv4/v6 Lookup Using Prefix Partitioning," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 1026–1039, July 2012, ISSN 0018-9340.

[7] M. Wang, S. Deering, T. Hain, and L. Dunn, "Non-random Generator for IPv6 Tables," in *Proc. of the 12th Annual IEEE Symposium on High Performance Interconnects, 2004*. IEEE Computer Society, Aug. 2004, pp. 35–40, ISBN 0-7803-8686-8.

[8] V. Pus, J. Tobola, V. Kosar, J. Kastil, and J. Korenek, "Netbench: Framework for Evaluation of Packet Processing Algorithms," in *Seventh ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS'11)*. IEEE Computer Society, Oct. 2011, pp. 95–96, ISBN 978-0-7695-4521-9.

A.4 Paper IV

Fast and Scalable Packet Classification Using Perfect Hash Functions

Fast and Scalable Packet Classification Using Perfect Hash Functions

Viktor Puš *
CESNET z. s. p. o.
Zikova 4, 160 00 Prague, Czech Republic
pus@liberouter.org

Jan Kořenek †
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
korenek@fit.vutbr.cz

ABSTRACT

Packet classification is an important operation for applications such as routers, firewalls or intrusion detection systems. Many algorithms and hardware architectures for packet classification have been created, but none of them can compete with the speed of TCAMs in the worst case. We propose new hardware-based algorithm for packet classification. The solution is based on problem decomposition and is aimed at the highest network speeds. A unique property of the algorithm is the constant time complexity in terms of external memory accesses. The algorithm performs exactly two external memory accesses to classify a packet. Using FPGA and one commodity SRAM chip, a throughput of 150 million packets per second can be achieved. This makes throughput of 48 Gbps for 40 B packet size. Further performance scaling is possible with more or faster SRAM chips.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays, Algorithms implemented in hardware*;
C.2.0 [**Computer-Communication Networks**]: General—*Security and protection (e.g., firewalls)*

General Terms

Design, Performance, Security

Keywords

Packet Classification, FPGA, SRAM

*This research has been partially supported by the Research Plan No. MSM, 6383917201 – Optical National Research Network and its New Applications
†This research has been partially supported by the Research Plan No. MSM, 0021630528 – Security-Oriented Research in Information Technology.

1. INTRODUCTION

With the rapid development of computer networks, traffic filtering has become one of the first steps in securing any network or computer. Basic traffic filtering device is the firewall, which makes per-packet decision based on the given set of rules. As network speeds are increasing, the demand for the speed of packet classification algorithms is also growing. Software solutions for the packet classification problem are available [3, 4], but their performance is not sufficient for wirespeed processing in the highest speed networks. Existing hardware approaches also do not fulfill performance requirements, or they require excessive amount of memory.

A classification algorithm contains a set of rules ordered by priority. Each rule defines a condition for all significant packet header fields. These fields are typically: Source IP Address, Destination IP Address, Source Port, Destination Port, Protocol. A condition may be exact match, prefix match (usually for IP addresses), range match (for ports), or a wildcard (matching any value). The goal of a packet classification algorithm is to find the matching rule with the highest priority. The output of the algorithm is then the number of the matched rule.

The traditional method of classifying packets makes use of Ternary Content Addressable Memories (TCAMs). However, the TCAM is an expensive device with high power-consumption [2]. It also matches only words with fixed data width and can limit throughput for complex rules. Therefore, algorithmic solutions without the use of TCAMs has become a research subject. While many algorithms have been published [7, 15, 17], none of them can match TCAM speed, because all existing algorithms require non-constant number of memory accesses in the worst case. This must be compared to the performance of TCAM solution, which classifies packet in a single memory access and the throughput is guaranteed. We propose a new packet classification method which uses SRAM to store necessary data, and FPGA to implement the algorithm. We will show that our solution is fully competitive to TCAM.

The rest of the paper is organized as follows: in the next section we discuss the related work and point out disadvantages of current solutions. Section 3 introduces a new packet classification algorithm. The most innovative part of the algorithm is described in detail in Section 4. Experimental results of our work are summed up in Section 5, and Section

6 concludes the paper. Finally, in Section 7 we discuss the possibilities of the future work in this area.

2. RELATED WORK

As the packet classification problem is inherently hard from a theoretical standpoint [7], a large number of hardware and software solutions [7, 15, 17] have been proposed. Solutions are based on exhaustive search, decision tree and grid-of-tries.

An interesting approach was introduced by Gupta in Hi-Cuts algorithm [14]. Hi-Cuts algorithm creates decision tree which cuts the packet space across one dimension at each level. The scheme was further improved by Hyper-Cuts [21] to cut the space across more dimensions at each level. In the Lucent bit vector scheme [17], range search is performed in each dimension, returning a vector with one bit for each rule. If one rule dimension is matched, its bit is set and a simple logical conjunction over all dimension vectors returns matching rules. Several improvements [8, 19, 22] were introduced later.

From the wide choice of available algorithms, we discuss only those which are related to our work. All of them belong to the family of decomposition-based methods. In decomposition methods, packet classification is divided into several steps. First step is the Longest Prefix Match (LPM) operation, which is performed independently in each dimension. From the given set of prefixes with various lengths, the LPM algorithm finds the one that best fits to the given full-length value. Range conditions (such as port ranges) in the ruleset are converted to prefixes, so that LPM may be performed in all dimensions.

LPM operation is performed in IP packet routing, so it is well studied topic. In fact, packet routing is a classification in one dimension only – the destination IP address. Basic algorithm for LPM is a trie, often modified to process more input bits in each step and to reduce memory consumption. Popular example of such algorithm is Tree Bitmap [12], but there are also many other solutions [10, 16, 18].

After LPM, all results must be combined together to get the resulting rule number. Basic Crossproduct algorithm [24] precomputes a crossproduct table, which contains resulting rule numbers for all possible combinations of prefixes. Because of the multiplicative nature of the crossproduct, this table may become extremely large. This table is implemented as a hash table, which yields issues with collisions. The whole crossproduct word must be stored in the table to detect a hash collision. In case a collision occurs, there must be a pointer to the next item. In this way, a linked list is created and performance may be reduced significantly.

Other method of combining LPM results together is the Distributed Crossproducting of Field Labels [25]. LPM is modified to return all valid prefixes (not only the longest one) for the given field value. What follows is the hierarchical structure of small crossproduct engines. Inputs of each engine are two sets of prefixes (or Labels, in general). Engine then performs set membership query for each possible pair of Labels. Result of the engine is another set of Labels. The result of the last engine is in fact a set of rules, from which the one with the highest priority is selected. Even when crossproducting is performed in a distributed way, it is still a weak point of the algorithm, because it is multiplicative in nature. If, for example, both input sets of the crossproducting engine have 10 items, then the engine has to perform $10 \times 10 = 100$ set membership queries.

Fast Packet Classification Using Bloom filters [11] brings further improvements to decomposition methods. The authors of this work replace crossproducts by *pseudorules*. To cover all valid combinations, certain rules are added to the ruleset. In fact, a pseudorule is always a special case of some rule. Example of pseudorules generation can be seen in Figure 1.

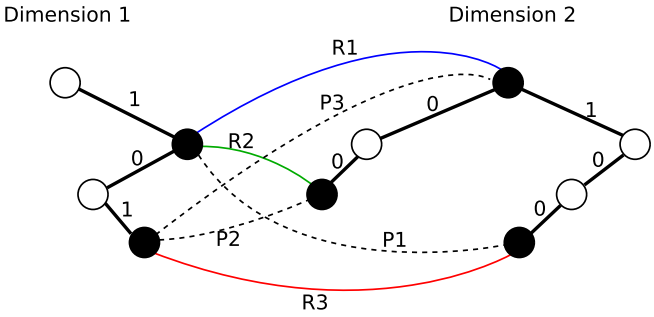

Figure 1: Three rules $R1, R2, R3$ and three added pseudorules.

Rule	Dimension 1	Dimension 2	Target rule
R1	1*	*	
R2	1*	00*	
R3	101	100	
P1	1*	100	R1
P2	101	00*	R2
P3	101	*	R1

Table 1: Rules and pseudorules.

We can see classification in two dimensions with three rules. For example, there is no rule for packet with header fields $(111, 100)$, but the correct result is rule $R1(1*, *)$[1]. Therefore pseudorule $P1(1*, 100)$ has to be added to cover this situation. Table 1 contains all rules and pseudorules together. *Target rule* in this table points to the correct classification result of pseudorule.

The generation of pseudorules has the character of crossproducting, and it may potentially expand the ruleset significantly, but not all possible combinations of prefixes need to be added. If the *universal rule* (a rule covering all possible packets) were in the ruleset, then all possible combinations would have to be added, but this rule can be removed from the ruleset and returned only if no other rule matches the packet.

Because pseudorules expansion is similar to crossproduct, the article provides heuristics on how to break ruleset into several subsets, eliminating the majority of pseudorules. The

[1]Symbol * denotes prefix or wildcard

paper also identifies rules that generate excessive amount of pseudorules. These rules are called *spoilers* and are removed to small on-chip TCAM. LPM operation is slightly modified to return result for each subset, because subsets may contain different prefixes. One Bloom filter is associated with each subset to perform set membership query. If the result is true, one Rule Table memory access is performed to retrieve resulting rule or pseudorule.

However, this scheme has several important disadvantages. Firstly, Rule Table is implemented using external SRAM, which imposes high requirements on SRAM throughput. If a rule format is very wide (for classification in more than five dimensions), the time required to read out one rule is also longer. We claim that Rule Table must be stored in an on-chip memory in order to achieve higher throughputs.

Secondly, an inherent property of the Bloom filter is nonzero probability of false positive errors. This may lead to a situation, when there exists a packet that causes false positives in several Bloom filters, resulting in several external memory accesses. If huge amount of such packets occurs in the network (e.g. during an attack), the classification algorithm slows down significantly.

Thirdly, Bloom filters are used only to reduce external memory accesses. Nevertheless, their implementation consumes on-chip resources which could be used in a more useful way.

Finally, the worst-case memory requirements are still exponential, even with the ruleset division into subsets and the use of TCAM for spoilers. But the algorithm does not try to reduce the size of one item – the whole rule or pseudorule has to be stored in an off-chip memory.

3. ALGORITHM

We propose a novel high-speed hardware-suited packet classification algorithm, which has a modular design and removes the drawbacks of Bloom filters, which were mentioned in Section 2. The algorithm consists of LPM for rule fields followed by a mechanism to search a rule. LPM and the rest of the classification algorithm have a very simple unidirectional interface and both parts may be changed separately, when a better solution is available. Recent research results for LPM operation have outstanding results even over 100 Gbps [18], therefore we do not propose any new architecture for LPM and focus on the rule searching mechanism.

With the bandwidth of the off-chip memory being the performance and scaling limitation for many existing solutions, we try to reduce amount of off-chip memory accesses for every incoming packet. Therefore, we propose to store the whole Rule Table in the on-chip memory using simple rule compression scheme and utilize the off-chip memory only to search the rule.

The primary goal is to find a solution with the constant packet rate and a good scalability with the size of the rule. Therefore, we propose to use a perfect hashing mechanism to provide the Rule Table search in a constant time and utilize the off-chip memory to store Perfect Hash Table.

The process of packet classification is divided into three basic steps (see Figure 2). The first step is the Longest Prefix Match operation, which is similar to approaches mentioned in Related work. The second step is mapping LPM results to the rule number, where we propose to use perfect hash function to perform fast searching. Even if the packet does not match any rule, the hash function will map the packet to some rule number. Because such invalid mapping can occur, it is necessary to include the third step, in which the packet is checked against the resulting rule. Therefore, the complete Rule Table has to be stored in the third step.

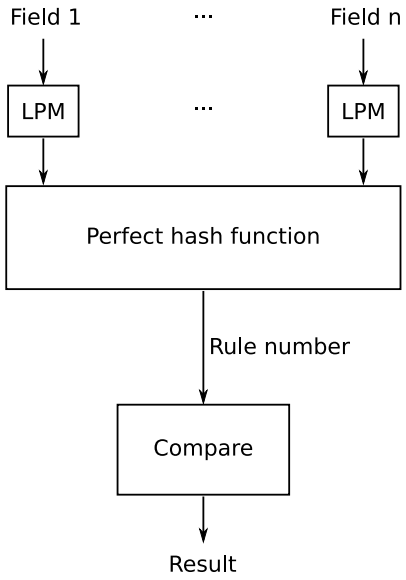

Figure 2: Three basic steps of the algorithm.

The perfect hash function must find a correct rule number for every packet. Thanks to LPM used in the first step, the packet state space is reduced significantly. The hash table is stored in the off-chip memory and its construction is described in the Section 4 of the paper.

The last part of the algorithm is the comparison of the rule to corresponding packet header fields. As high throughput memory is needed to read a rule, the Rule Table is stored in on-chip memories. The on-chip memory has a limited capacity, therefore all rules are compressed to save as many memory resources as possible. We propose simple prefix indexing scheme (see Figure 3) to reduce the Rule Table size significantly. The rule itself contains only several indexes to adjacent Prefix Tables, where all prefixes are stored. Port number is stored directly in the Rule Table, because it is a small field. This exploits the property of Rule Tables we and others [25] have observed: the number of unique prefixes in each dimension is usually quite small, therefore Prefix Tables will be also small. The experimental results for rulesets mentioned later in the text show that memory was reduced at least by one half, compared to simple direct storage of rules.

This compression makes use of hardware parallelism, because all Prefix Tables are accessed in the same time. Off-

chip Rule Table implementation cannot be fast enough when using this scheme.

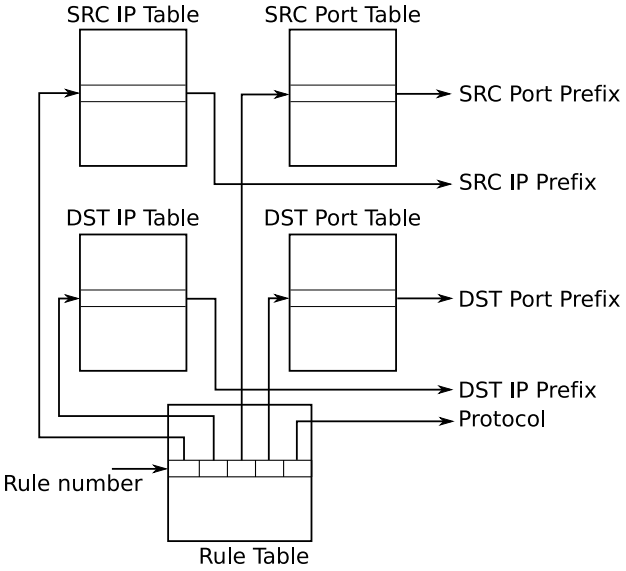

Figure 3: Prefix indexing scheme to reduce the Rule Table size.

4. PERFECT HASH FUNCTION

The problem of designing the perfect hash function could be described as follows: each independent LPM returns one word for every packet. Each word represents one prefix and all prefixes together have a meaning of one (possible) pseudorule. Each pseudorule is associated with one rule. We seek a function mapping all valid pseudorules to their associated rules. Invalid pseudorules (for a packet that matches no rule) may be mapped to any rule, because this false positive is resolved later by simple comparison.

We chose static perfect hashing, because dynamic schemes have significantly greater overhead. Instead of dynamic rules insertion or removal, we can simply recompute the whole static perfect hash. We propose to use a perfect hash construction algorithm described in [9] to get a hash function, which for each pseudorule returns a number of its associated rule. From a wide choice of static perfect hashing schemes, we chose this one because of its simplicity and good results. Perfect hashing has been proposed to be used in networks applications before [20, 13, 23, 6], but according to our knowledge, the idea of using perfect hash functions is novel in the field of multidimensional packet classification.

The perfect hash construction algorithm creates acyclic graph, where edges are the keys, and vertices are results of two different hash functions. Vertices are then assigned values so that they sum up to the desired hash value. Detailed description can be found in [9]. The algorithm consists of seven basic steps:

1. Input: K keys, each associated with a number which it is to be hashed to.
2. Create graph with $N = cK$ vertices, where $c > 1$.
3. Pick any two different ordinary hash functions f_1, f_2 that output values $0 \ldots N-1$.
4. For each key, compute $h_1 = f_1(key), h_2 = f_2(key)$, draw an edge between vertices h_1 and h_2 of the graph and associate the desired hash value with that edge.
5. Check if the graph is acyclic. If not, increase c and go to step 2.
6. Associate values to each vertex such that for each edge you can add the values of both its vertices and get the desired value for the edge. This may be done by depth-first search algorithm, because the graph is acyclic.
7. f_1, f_2 and vertex values now make up the desired function.

In our algorithm, keys are rules and pseudorules in the form of concatenated LPM results, and associated numbers are numbers of the correct rule. This way, we get a function that hashes rule and all its associated pseudorules directly to the correct rule number. In fact, we introduce intended collisions of the hash function. The idea of intended hash collisions is a non-traditional usage of perfect hash functions. The important point is that none of pseudorules is stored in our scheme. Therefore, way we save significant amount of memory.

Table 2 and Figures 4 and 5 show how a graph for the example in Table 1 could look like. When the graph is created, the hash function is simple. At first, two different hash functions are evaluated over the input word. Then two vertex values are read from the Vertex Table and added. For each vertex, only one integer is stored.

Input word	f1	f2
<1*, *>	0	7
<1*, 00*>	6	0
<101, 100>	5	4
<1*, 100>	0	4
<101, 00*>	1	3
<101, *>	3	2

Table 2: Two hypothetical hash functions' results for inputs in the form of encoded and concatenated LPM results

By theory, acyclic graph with n edges must have at least $n + 1$ vertices. This means that a table with more items than the number of rules and pseudorules is needed. The perfect hash algorithm usually needs greater overhead. Our experiments in Section 5 show that the table size must be approximately twice the theoretical minimum, which is good result among other perfect hash algorithms.

Similarly to [11], we use small on-chip TCAM to store spoilers and save significant amount of memory. Identifying the greatest spoilers is a complex task, which needs to be further investigated. For our experiments, we use semi-automatic method, and we intend to do more research in finding an automatic heuristic method with good results.

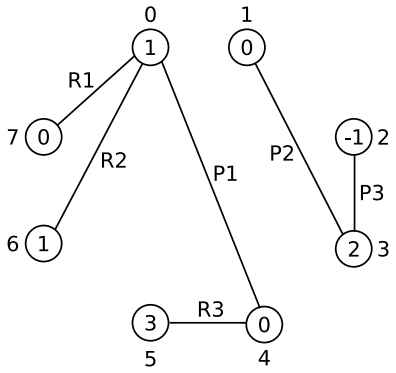

Figure 4: Example graph with 6 (pseudo)rules and 8 vertices.

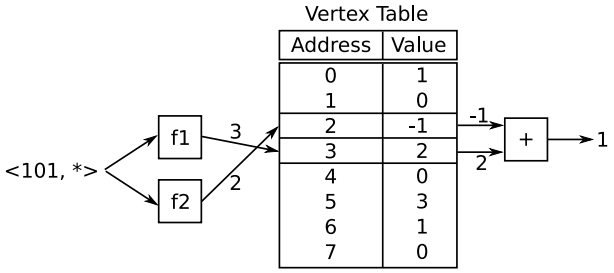

Figure 5: Example of computing perfect hash function.

5. RESULTS

The proposed algorithm was implemented in FPGA and utilizes one external static memory. We have studied several rulesets to get information about typical properties of rules. Similarly to [25], we have found that the number of unique prefixes in each dimension is usually quite small (see Table 3). Therefore, the data structures for LPMs may be easily stored in small on-chip memories, either BlockRAMs or distributed memories. Also the Rule Table itself is not greater than a few kilobytes; therefore, we do not need an external memory for its storage.

The table of graph vertices may become large, therefore, we propose to use an external memory to store the Perfect Hash Table. On-chip memories could be used only in case of small rulesets. For example, Xilinx FPGA Virtex5 LX110 [5] contains 4 608 kb of BlockRAM memory, which gives us 262 144 vertices (suppose 18 bits for one vertex).

The proposed solution significantly reduces bottleneck caused by the speed of external memory. Only two 18-bit words need to be read for every packet, which is many times less than one whole rule (or even worse, several rules) [11]. Moreover, performance of our algorithm is not affected by the complexity of rules. Other fields (i.e. MAC addresses, TCP flags, etc.) can be added to rules and the throughput remains the same, only on-chip Rule Table size increases linearly. It means that our solution scales well with the rule size.

Ruleset	Rules	SRC Addr	DST Addr	SRC Port	DST Port	Protocol
fw1	32	13	2	10	22	4
fw2	58	26	24	4	1	2
fw3	103	28	48	36	1	4
fw4	171	84	84	1	6	3
synth1	40	12	19	10	22	5
synth2	49	35	41	8	22	3
synth3	49	26	14	14	1	4
synth4	70	27	62	1	48	3
synth5	82	20	37	3	3	4
synth6	100	73	85	1	54	4

Table 3: Numbers of unique prefixes in each dimension.

5.1 Performance

Similarly to [11], we suppose 300 MHz DDR memory with the burst length of two words. The throughput of our solution is compared to the Crossproduct algorithm and Bloom filters in Table 4. We do not take into account the speed of LPM operation, because we consider it fast enough [18]. It can be seen that our solution has a constant throughput and does not require very wide external memory data bus. The time to recompute all necessary data structures was always below 4 seconds. We use two Jenkins hash functions [1] with various seeds to implement the perfrect hash function.

Data Width	Crossproduct Based	Bloom Filter-Based 4	6	8	Perfect Hash
9	37.5	9.375	6.25	4.6875	150
18	75	18.75	12.5	9.375	150
36	150	37.5	25	18.75	150
72	300	75	50	37.5	150

Table 4: Throughput (in millions packets per second) for several data bus widths of 300 MHz DDR memory. Rule word width of 144 bits is considered. For the algorithm exploiting Bloom filters we consider three cases: match of four, six and eight rules for each packet.

5.2 Memory requirements

We performed pseudorules expansion and perfect hash function search for several rulesets from university campus network (fw) together with several synthetic ruleset generated by ClassBench [26] (synth) to determine off-chip memory requirements. Memory requirements are compared to Bloom filters-based and Crossproduct algorithm in Table 5.

As can be seen, numbers of graph vertices may become prohibitive for on-chip memories, but is acceptable for commodity SRAM chips. Each vertex is stored as one signed integer, actual range of vertex values determines number of bits required to represent it. If SRAM works with larger data width, words can be split into several parts to multiply the available table size.

Ruleset	Rules	Crossprod.	Bloom F.	Perf. Hash
fw1	32	3 618	823	740
fw1	58	13 086	1 492	3 424
fw3	103	1 008 954	2 651	252 220
fw4	171	443 484	4 401	116 356
synth1	40	11 070	1 029	2 740
synth2	49	29 520	1 261	6 601
synth3	49	19 278	1 261	5 035
synth4	70	10 512	1 801	2 451
synth5	82	90 324	2 110	22 495
synth6	100	17 010	2 574	3 827

Table 5: Off-chip memory requirements (in Bytes) for several rulesets. For Bloom Filter-Based algorithm we assume that pseudorules expand the ruleset by the factor of 1.43 (average from the original paper [10]). For Crossproduct and Perfect Hash scheme we use on-chip TCAM for 16 spoilers.

Overall chip area is hard to compare to other solutions, because every implementation has many variable parameters (speed, number of stored prefixes, selection of classification dimensions). However, we provide informal comparison to [11]:

- Both schemes use LPM as the first step.
- In our solution, we need only two various on-chip hash functions to compute the perfect hash function, while [11] uses many hash functions to implement Bloom filters.
- Our solution stores the ruleset in on-chip memories.
- In [11], small bit array is stored for each Bloom filter.
- Both schemes use external memory and other common blocks (packet recieve and transmit modules etc.).

To verify our results, we have implemented the described algorithm for the Virtex 5 LX110T FPGA. We used 125 MHz working frequency and we set the throughput to two cycles per packet. This limitation of the particular implementation is induced by our current needs – we have connected two 10 Gbps network interfaces and one PCI-Express x8 bus to the FPGA.

We added four other classification dimensions: Source and Destination MAC address, TCP flags and Input interface number. We were able to load up to 1 000 rules into the device. Data structures (LPMs, Vertex Table, etc.) generation time was below 0.5 second on a PC with 2 GHz Intel Pentium processor. This is also the update delay if the ruleset changes.

Using the proposed algorithm, we have created two-port firewall with the constant aggregated throughput of 62.5 million packets per second.

6. CONCLUSION

We have proposed a novel algorithm for fast packet classification using perfect hash functions. Our algorithm introduces intended hash collisions to reduce memory requirements. By creating custom hash function, we make sure that all pseudorules are hashed to associated rule, which means that no pseudorule has to be stored in the memory and significantly less memory is needed. The results in Section 5 show that the only larger amount of memory is utilized to store Perfect Hash Table, even for large rulesets.

Because only two external memory accesses are needed to classify a packet, 150 million packets per second can be processed with commodity FPGA and SRAM. This packet rate corresponds to 100 Gbps Ethernet for the shortest packets. Moreover, the throughput doesn't depend on ruleset complexity and is well scalable with number of external memories.

According to our knowledge, the proposed algorithm is the first algorithm which requires reasonable amount of memory and has constant processing time even for complex ruleset. High throughput together with constant processing time makes the proposed algorithm fully competitive to widely used TCAM solutions. As the proposed solution uses commodity SRAM, the price and power consumption is significantly lower than classification with TCAM memory.

7. FUTURE WORK

We continue to explore this method to further improve memory efficiency by reducing size of the Vertex Table for large rulesets. If the memory requirements drop under certain limit, only on-chip memory can be used to store the Vertex Table and the classification process can be significantly faster. Moreover, if external memory is removed, the price and power consumption is decreased. We also believe that the idea of intended hash collisions has potential value for other tasks which allow relatively slow precomputation, but require extremely fast search times.

8. REFERENCES

[1] A hash function for hash table lookup. http://burtleburtle.net/bob/hash/doobs.html, December 2008.

[2] IDT Generic Part: 75K72100. http://www.idt.com/?catID=58523&genID=75K72100 , June 2008.

[3] Netfilter: firewalling, NAT and packet managing for Linux. http://www.netfilter.org/ , June 2008.

[4] PF: The OpenBSD Packet Filter. http://www.openbsd.org/faq/pf/ , June 2008.

[5] Xilinx Virtex–5 Family FPGAs. Xilinx, Inc.

[6] N. S. Artan and H. J. Chao. Tribica: Trie bitmap content analyzer for high-speed network intrusion detection. *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 125–133, May 2007.

[7] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to CAMs? In *INFOCOM*, 2003.

[8] F. Baboescu and G. Varghese. Scalable packet classification. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 199–210, New York, NY, USA, 2001. ACM.

[9] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.

[10] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using Bloom filters. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 201–212, New York, NY, USA, 2003. ACM.

[11] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. Fast packet classification using Bloom filters. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 61–70, New York, NY, USA, 2006. ACM.

[12] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[13] S. Giordano, F. Oppedisano, G. Procissi, and F. Russo. A novel high-speed micro-flows classification algorithm based on perfect hashing and direct addressing. *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pages 448–452, November 2007.

[14] P. Gupta and N. McKeown. Packet classification using hierarchical intelligent cuttings. In *Proc. Hot Interconnects*, 1999.

[15] P. Gupta and N. McKeown. Algorithms for packet classification, 2001.

[16] P. Gupta, B. Prabhakar, and S. P. Boyd. Near optimal routing lookups with bounded worst case performance. In *INFOCOM*, pages 1184–1192, 2000.

[17] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.*, 28(4):203–214, 1998.

[18] H. Lee, W. Jiang, and V. K. Prasanna. Scalable High-Throughput SRAM-Based Architecture for IP Lookup Using FPGA. In *FPL '08*. IEEE, 2008.

[19] J. Li, H. Liu, and K. Sollins. AFBV: a scalable packet classification algorithm. *SIGCOMM Comput. Commun. Rev.*, 32(3):24–24, 2002.

[20] Y. Lu, B. Prabhakar, and F. Bonomi. Perfect hashing for network applications. *Information Theory, 2006 IEEE International Symposium on*, pages 2774–2778, July 2006.

[21] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 213–224, New York, NY, USA, 2003. ACM.

[22] H. Song and J. W. Lockwood. Efficient packet classification for network intrusion detection using FPGA. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 238–245, New York, NY, USA, 2005. ACM.

[23] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis. A reconfigurable perfect-hashing scheme for packet inspection. *Field Programmable Logic and Applications, 2005. International Conference on*, pages 644–647, Aug. 2005.

[24] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.*, 28(4):191–202, 1998.

[25] D. Taylor and J. Turner. Scalable packet classification using distributed crossproducting of field labels. In *IEEE INFOCOM 2005, 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, pages 269–280, July 2005.

[26] D. E. Taylor and J. S. Turner. Classbench: a packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, 2007.

A.5 Paper V

High-speed Regular Expression Matching with Pipelined Automata

High-speed Regular Expression Matching with Pipelined Automata

Denis Matoušek
Faculty of Information Technology
Brno University of Technology
Božetěchova 1/2, 61266 Brno
Czech Republic
Email: imatousekd@fit.vutbr.cz

Jan Kořenek
and Viktor Puš
CESNET a.l.e.
Zikova 4, 16000 Prague
Czech Republic
Email: korenek,pus@cesnet.cz

***Abstract*—Pattern matching is a complex task which is widely used in network security monitoring applications. With the growing speed of network links, pattern matching architectures have to be improved in order to retain wire-speed processing. Multi-striding is a well-known technique on how to increase throughput of pattern matching architectures. In the paper we provide an analysis of scalability of multi-striding and show that it does not scale well and cannot be used for 100 Gbps throughput because utilization of FPGA resources grows exponentially. Therefore, we have designed a new hardware architecture for high-speed pattern matching that combines the multi-striding technique and parallel processing using pipelined finite state machines (FSMs). The architecture shares a single packet buffer for all parallel FSMs. Efficient implementation of the packet buffer reduces the number of BlockRAMs to 18% when compared to simple parallel implementation. Instead of multiplexing input data, the architecture pipelines the states of FSMs. Such pipelined processing with only local communication has a direct positive impact on frequency and throughput and allows us to scale the architecture to hundreds of Gbps.**

I. Introduction

Regular Expression (RE) matching is a widely used operation in computer networks for the identification of application protocols, detection of network attacks, application-aware load balancing and for many other network applications. Current processors are not powerful enough to match REs at 10, 40, or 100 Gbps speed [1]. In order to achieve high throughput, some form of hardware acceleration is needed.

Many hardware architectures have been designed to accelerate pattern matching for Intrusion Detection Systems (IDS) [2]–[8] where network traffic has to be matched against thousands of REs. Several architectures take advantage of massive parallel processing in Field Programmable Gate Arrays (FPGAs) and use mapping of Non-deterministic Finite Automata (NFA) to FPGA [2]–[5], [9]. In order to achieve linear time complexity, all non-deterministic paths are processed in FPGA simultaneously. As the number of REs in IDS systems increases in time, many optimizations have been introduced to reduce FPGA logic utilization and map more REs to FPGA [4], [10]. Most of these optimizations are only focused on REs in IDS systems Snort [11] and Bro [12].

978-1-5090-5602-6/16/$31.00

Prasanna has introduced modular RE-NFA [13], [14], which can be converted automatically into a modular circuit on the FPGA. The circuit can be created from a set of REs without synthesis and can be uploaded to the FPGA by dynamic reconfiguration. Dynamic reconfiguration is also used for a fast update of the RE set in Dynamic BP-NFA [15].

The architectures based on Deterministic Finite Automaton (DFA) [6]–[8] use memory to store the transition table, which enables an even faster update of the pattern set. FPGA reconfiguration is not needed because the architecture remains the same. Only memory content is updated if the RE set is changed. On the other hand, the construction of DFA from NFA has exponential time complexity and can cause an exponential growth of states and transition table, which has a direct impact on memory requirements. Therefore, memory requirements have been reduced by Delayed input DFA [6], compression [7], [16] and other optimizations [8], [17]. Other architectures use a combination of NFA and DFA [18]–[20] to cope with large data sets and exponential growth of DFA states caused by .* constructions in REs.

Most hardware architectures provide a reduction of memory requirements or FPGA logic utilization to match more REs. It is also necessary to increase the matching speed with the growing speed of network links. For pattern matching, the speed depends on frequency and the number of input symbols (bytes) processed per clock cycle. As the FPGA frequency increases only slightly over time, it is necessary to process more input symbols at once.

Brodie has presented the first architecture with accepting multiple input symbols per clock cycle [21]. Prasanna has introduced spatial stacking for multi-character matching [13] with RE-NFA, but high fanout of the final circuit significantly decreases the frequency, even when matching only eight input symbols per clock cycle. It is important to note that more than 64 input symbols have to be processed at once in order to achieve 100 Gbps throughput. Becchi introduced general multi-striding technique [22], which can be used for NFA or DFA automata and is widely used to increase throughput of RE matching architectures.

Multi-striding causes high memory requirements in DFAs. Therefore, we focus on NFA architectures, where multi-

striding is used to increase processing speed at the cost of FPGA logic utilization. We have performed an analysis on how the amount of FPGA logic grows with the processing speed. The analysis shows that multi-striding significantly decreases frequency and causes exponential growth of utilization of FPGA logic similar to spatial stacking. Therefore, we propose a new hardware RE matching architecture for high-speed network links with the throughput over 100 Gbps. The architecture uses pipelined automata directly mapped to the FPGA logic and is well scalable. The throughput can be easily increased by the number of automata in the pipeline at the cost of only the linear growth of utilization of FPGA resources. Moreover, the architecture uses a single input packet buffer to reduce memory requirements.

The work is divided into six sections. The introduction is followed by an analysis of multi-striding in Sec. II. Sec. III discusses parallel architectures. Its subsection III-A describes features of parallel architectures and subsection III-B presents the new high-speed architecture. The evaluation and results are presented in Sec. IV and Sec. V followed by the conclusion in Sec. VI.

II. Analysis of Multi-striding

The purpose of this section is to analyze whether multi-striding is scalable to achieve the throughput of 100 Gbps and more. *Multi-striding* is a technique to increase throughput of a system for matching strings against a set of regular expressions [22]. It is used to convert a finite automaton into a modified one that is equivalent in terms of accepted language. The difference is that input symbols of the modified finite automaton are made up by concatenation of several input symbols of the original finite automaton so that more input symbols of the original finite automaton can be processed in a single step. The number of input symbols processed at once is called *level of multi-striding*.

We selected NFA-based architecture [3] for the analysis as it can be easily mapped onto FPGA logic and it generally achieves a high frequency. In NFA-based architecture, transitions are mapped to LUT (look-up table) elements of an FPGA chip. Multi-striding increases the number of transitions and, moreover, many transitions can be connected to a single state. Then, next-state logic is more complex, more LUTs are on the critical path and maximal achievable frequency is lower. Therefore, we have inspected how FPGA utilization and maximal achievable frequency changes with the level of multi-striding.

We compared two scenarios. The first one is processing with a single multi-striding automaton with the level of multi-striding equal to the width on input data bus (expressed as the number of symbols). The second one is processing with multiple parallel multi-striding automata. Each of these automata processes a portion of the input data bus and has reduced level of multi-striding compared to the first case. We used automata with the level of multi-striding ranging from 1 to 64. It is important to note that the analysis is focused only on the scalability of multi-striding. The overhead caused by the distribution of input data among multiple parallel automata is not considered.

The analysis was performed by a set of rules from the L7 classifier for Linux Netfilter that are marked "Great: Works."[1] Synthesis was performed for Xilinx Virtex-7 VH580T chip using Vivado tool v.2016.1.

The results of the analysis are shown in the graph in Fig. 1. The graph shows the scalability of multi-striding and parallel configurations in terms of FPGA logic utilization and achievable throughput. Throughput is based on maximal achievable frequency of a specific configuration and input data bus width. Each data point represents the required number of LUT elements on the y axis that are necessary to process data stream with throughput on the x axis. Each data series is labelled *parallel xN*, except for the data series of a single multi-striding automaton, and represents a system comprised of multiple multi-striding automata processing N input symbols at once. For example data series labelled *parallel x16* shows an achievable throughput of a system comprised of multiple multi-striding automata, each processing 16 input symbols at once. The more multi-striding automata which are necessary so as to achieve the required throughput, the more LUT elements are used.

In the graph in Fig. 1 we can see the exponential growth in utilization of LUT elements with the level of multi-striding. On the other hand, utilization of LUT elements increases only linearly with the number of parallel multi-striding automata.

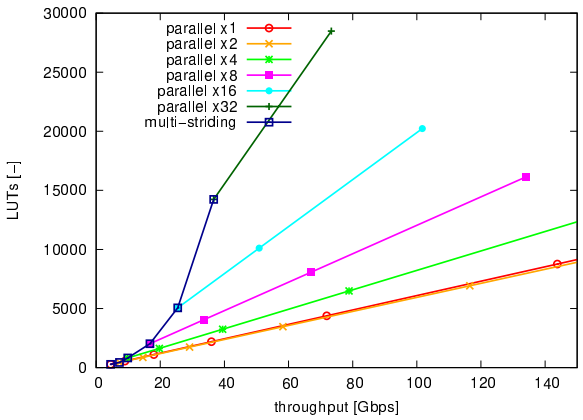

Fig. 1. Throughput of a single multi-striding automaton (*multi-striding*) and parallel configurations (*parallel xN*)

The graph in Fig. 1 is based on the values from Table I and Table II. Table I shows maximal achievable throughput for various levels of multi-striding and parallel configurations. The rows correspond to the levels of multi-striding and the columns correspond to the widths of input data buses (expressed as the number of symbols). The first column (marked †) represents the level of multi-striding of each single automaton. The second column (marked ‡) shows maximal frequency in MHz, the other table cells show throughput in Gbps. Let's consider input data bus that is 16 symbols wide. The corresponding

[1] http://l7-filter.sourceforge.net/protocols

column is marked with the number *16*. The case of the first row labelled as *1* results in a system comprised of 16 parallel automata, each processing one input symbol at a time. The case of the second row labelled as *2* results in a system comprised of eight parallel automata using multi-striding technique to process two input symbols at once, etc. If the level of multi-striding is greater than the width of input data bus, the corresponding cell contains a value *N/A* (not applicable).

Table II shows the utilization in terms of LUT elements. The table is organized in the same way as Table I. The rows correspond to the levels of multi-striding and the columns correspond to the widths of input data buses (expressed as the number of symbols). We can see that the most suitable level of multi-striding is four since it results in the lowest utilization of LUT elements (not considering the first two columns).

It is important to note that we were not able to synthesize automata for the level of multi-striding greater than 32. The synthesis failed due to exhaustion of system resources. Therefore, maximal frequency, throughput, and FPGA logic utilization are presented only up to this level of multi-striding.

TABLE I
THROUGHPUT IN GBPS FOR VARIOUS CONFIGURATIONS OF PARALLEL AUTOMATA

†	‡	Width of input data bus						
		1	2	4	8	16	32	64
1	561	4.4	8.9	17.9	35.9	71.9	143.8	287.6
2	455	N/A	7.2	14.5	29.1	58.2	116.5	233.0
4	308	N/A	N/A	9.8	19.7	39.4	78.8	157.7
8	261	N/A	N/A	N/A	16.7	33.5	67.0	134.0
16	198	N/A	N/A	N/A	N/A	25.4	50.8	101.7
32	143	N/A	N/A	N/A	N/A	N/A	36.6	73.3
† Level of multi-striding of parallel automata ‡ Max. frequency [MHz]								

TABLE II
NUMBER OF LUT ELEMENTS FOR VARIOUS CONFIGURATIONS OF PARALLEL AUTOMATA

†	Width of input data bus						
	1	2	4	8	16	32	64
1	274	548	1096	2192	4384	8768	17536
2	N/A	433	866	1732	3464	6928	13856
4	N/A	N/A	811	1622	3244	6488	12976
8	N/A	N/A	N/A	2018	4036	8072	16144
16	N/A	N/A	N/A	N/A	5057	10114	20228
32	N/A	N/A	N/A	N/A	N/A	14239	28478
† Level of multi-striding of parallel automata							

The second column (marked ‡) of Table I reveals that the frequency of multi-striding automata rapidly decreases as the level of multi-striding increases. The data series marked *multi-striding* of the graph in Fig. 1 shows that the FPGA logic utilization of multi-striding automata increases exponentially with the throughput. It also shows that maximal achievable throughput is only 36 Gbps, even for NFA-based architecture that is well suitable for synthesis tools.

From the analysis we can see that multi-striding is not scalable and cannot be used to achieve throughput in the order of hundreds of gigabits. For high-speed networks, it is better to use multiple multi-striding automata in parallel. Therefore, we analyzed several approaches using parallel automata and designed a new scalable architecture.

III. ARCHITECTURE

A. Architectures with Parallel Automata

We have considered several architectures to cover the approach described in the previous section. The most straightforward is the architecture depicted in Fig. 2.

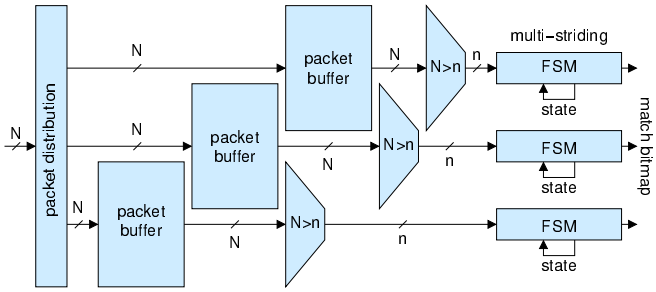

Fig. 2. Parallel architecture with distributed buffers

It is based on the distribution of incoming packets to several separate data paths, where packets are processed at a lower speed. Input data width is N and each data path contains an NFA that processes n bits ($n < N$) using multi-striding. The drawback of this architecture is the necessity of data width transformation of each data path. In order to guarantee the processing of packets without wait states, it is necessary to place a queue in front of each data width transformer. The queue will compensate for lower speed of data processing in NFAs. The queues are implemented with block RAMs of the FPGA chip because the size of queues is too large. If implemented in distributed memory, they would consume a lot of LUTs. Block RAMs of Virtex-7 chip have a capacity of 32 Kbits (without parity part) and a read/write port with a configurable width. Maximal width of a read/write port is 64 bits (eight parity bits are not used). For example, in order to store 512 bits wide items, we need eight such block RAMs. Minimal capacity of a queue comprised of eight block RAMs is then $8 \times 32\,\mathrm{Kbits} = 256\,\mathrm{Kbits} = 32\,\mathrm{KB}$. Given an Ethernet frame of a maximum length of 1518B (Jumbo frames are not considered), utilization of used block RAMs is only 4.63%, which indicates very low memory effectiveness of this approach.

Low effectiveness of memory utilization can be removed with a shared packet buffer for all NFAs, which is shown in Fig. 3.

The architecture uses parallel NFAs with multi-striding. Each NFA processes n bits at once in the same way as the

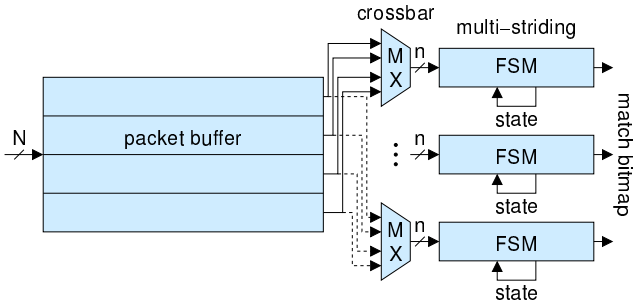

Fig. 3. Parallel architecture with a shared packet buffer and large multiplexers

previous architecture. The drawback of this approach is the necessity of placing a multiplexer in front of each NFA, basically forming a crossbar. The multiplexers allow for reading any part of the input data from the packet buffer. The wider the input data is, the more multiplexers there are, and it is more complicated for the synthesis tool to route interconnections among multiplexers, shared decoders and NFAs. In order to analyze the suitability of this architecture, we have focused on the multiplexer network itself and synthesized it in several configurations of data width. Table III shows LUT utilization. *Input width* column represents the width of input data bus and *paths width* columns represent the width of NFA data paths. *MUX* columns represent the number of multiplexers that are necessary for corresponding configurations. Table III shows that LUT utilization of the multiplexer network itself becomes unacceptable for wider input data.

TABLE III
LUT UTILIZATION OF MULTIPLEXER ARCHITECTURE

paths width	8b		16b		32b	
input width	MUX	LUT	MUX	LUT	MUX	LUT
64	8	128	4	64	2	32
128	16	512	8	256	4	128
256	32	2304	16	1024	8	512
512	64	18786	32	8352	16	6064
1024	128	78951	64	37120	32	16984
2048	256	310843	128	166988	64	61760

The common drawback of the two above mentioned approaches is the necessity of transfer and/or storage of a large amount of data among the components. The bigger the amount is, the more difficult it is to place and route the circuit for the synthesis tool. It implies lower frequency of the circuit.

B. Architecture with Pipelined Automata

We propose an architecture depicted in Fig. 4. It is based on the idea of direct connection of automata to the packet buffer without a complex multiplexer network. The packet buffer is shared for all automata that perform matching against a set of regular expressions. Automata are interconnected so that they can pass their state from one to another in a successive way. We call this concept *pipelined automata*. Looking at Fig. 4, the first part of the packet P_1 is processed in FSM_1. The second part of the packet P_1 is processed in FSM_2 based on the state from FSM_1. FSM_1 is then ready to process another packet.

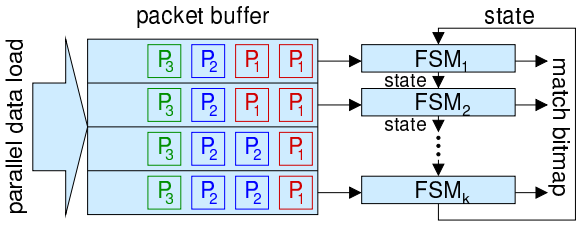

Fig. 4. Processing of packets in pipelined automata

A more detailed description of processing parts of packets in time is depicted in Fig. 5. The first row corresponds to the first pipelined automaton, the second row corresponds to the second pipelined automaton, etc. All automata (FSM) are the same. Processing of packet P_1 starts in FSM_1 where the first data word is used to compute the next state. The result is passed to FSM_2, which uses it as current state and computes the next state from the next data word. Then, the result is again sent to the next pipeline stage (the next FSM). This way, the automata process the packet P_1 word by word until the whole packet is processed. The pipeline is fully utilized and has maximal throughput if k packets are processed by k FSMs.

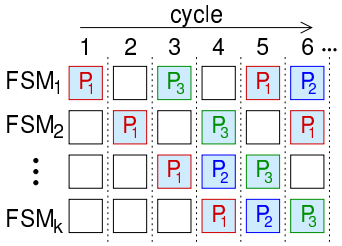

Fig. 5. Processing of packets in the pipeline

We will now describe the pipelined automata in more detail. Pipelined automata use the vertical arrangement of automata with the same number of states as the original automaton. Let's denote F the original automaton, F_1 the first pipelined automaton and F_k the last pipelined automaton. Then, the transition p of the automaton F from the state s_i to the state s_j is transformed into k transitions in the pipelined automata as follows: for $y = 1, \ldots, k-1$, the resulting transition leads from the state s_i in the automaton F_y to the state s_j in the automaton F_{y+1}. For $y = k$, the resulting transition leads from the state s_i in the automaton F_y to the state s_j in the automaton F_1. Good data locality can be achieved using appropriate data distribution among pipelined automata.

It should be noted that all automata $F_i, i = 1, ..., k-1$ have the same form as the original automaton F. The only modification is the passing of the state among the automata. Thus, a pipelined automata does not limit expressiveness

in any way and has the same expressive power as a finite automata.

An example of pipelined automata is shown in Fig. 6. The left part a) shows original automaton F and its transitions. The right part b) shows corresponding pipelined automata made up of the automata F_1, F_2, and F_3. Pipeline stages are divided by vertical dot lines. The transition of the original automaton shown in red is transformed into three transitions leading from the states of the first pipeline stage in the pipelined automata. The transitions that overlap from the last pipelined automaton to the first one are shown as dashed. The states s_1, s_2, s_3, and corresponding transitions are colored and they represent a pass in the original automaton and corresponding pipelined automata.

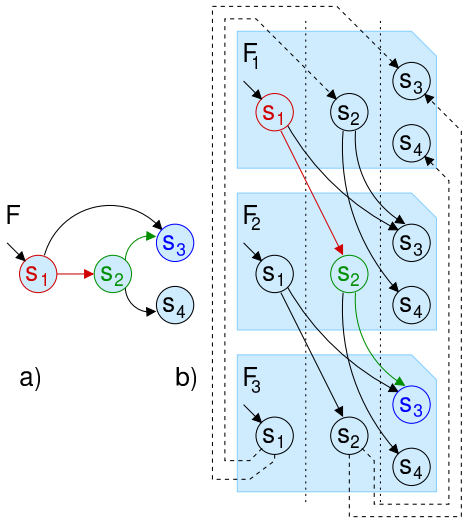

Fig. 6. Pipelined automata (b) and the original automaton (a)

The detailed architecture of the system is depicted in Fig. 7. Block RAMs of the packet buffer are arranged one above another. Input data are split into blocks and distributed in a round-robin way among block RAMs. The size of blocks and the width of read/write ports of block RAMs are configured based on the level of multi-striding used in the automata. Blocks of input data are written to block RAMs by a shared component marked as *distribution*. Each block RAM has a *memory control* component attached. The *memory control* components pass the data from block RAMs to the corresponding automata. The *memory control* components are interconnected for the purpose of the synchronization of sequential readings of the blocks of a single packet. The components *request buffer* and *reservation unit* keep start addresses and lengths of the packets stored in block RAMs. Fig. 7 shows the processing of the first four blocks of a packet in pipelined automata. Assignment of pattern matching results to corresponding packets is not shown for reasons of simplicity.

IV. Evaluation

The evaluation was focused on 100G Ethernet standard that is currently the fastest approved standard of Ethernet.

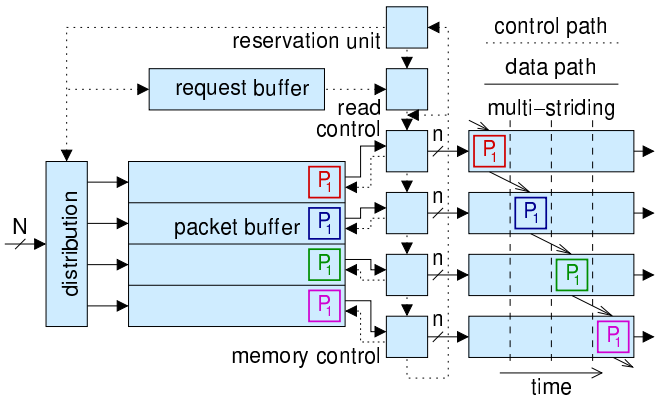

Fig. 7. Hardware architecture with pipelined automata

100G Ethernet-based standards (100GBASE-*) define variants SR10, LR4, and SR4 for optical links. The first one works with ten 10 Gbps channels and LR4 and SR4 work with four 25 Gbps channels. The evaluation was carried out by synthesizing the system for Xilinx Virtex-7 VH580T chip that contains GTZ transceivers that are necessary to process high-frequency 25 Gbps signals.

Synthesis was performed using the NetCOPE development framework[2], which provides underlying components and build system for the development of firmwares for FPGA chips. The frequency is set to 200 MHz because of complex components for network communication and communication with the host machine (DMA engine for PCIe bridge). We can deduce the width of data buses based on required throughput and considered frequency. The widths in Table IV are rounded up to be a power of two.

TABLE IV
Widths of data buses

Required throughput	Data bus width	Number of input symbols
10 Gbps	64b	8
40 Gbps	256b	32
100 Gbps	512b	64
400 Gbps	2048b	256

We chose three sets of regular expressions for evaluation:

- A set of rules from the L7 classifier for Linux Netfilter that are marked as "Great: Works."[3] (12 regular expressions). This is denoted *L7 great* in the following text.
- A set of rules of the backdoor module of Snort application[4] (154 regular expressions). This is denoted *Snort* in the following text.
- Our own set of six rules to detect four basic L7 protocols (HTTP request/response, SIP request/response, DNS, SMTP). This is denoted *L7 selected* in the following text.

[2] https://www.liberouter.org/technologies/netcope/
[3] http://l7-filter.sourceforge.net/protocols
[4] https://www.snort.org/downloads/#rule-downloads

We compared three approaches based on our own reference implementation in VHDL language. The first one uses multi-striding to process multiple input symbols at once. The advantage of this approach is that no block RAMs are needed. However, transformation of the original automaton to a version processing several input symbols at once turned out to be an issue due to space complexity of the algorithm as the amount of transitions of the automaton grows exponentially. The most complex automaton we were able to generate was for the L7 great set for throughput of 40 Gbps (32 input symbols processed at once).

The second approach uses parallel architecture that distributes packets to several separate data paths. The third approach uses our proposed architecture with a shared buffer and pipelined automata. We decided to use pipelined automata with the level of multi-striding equal to four (four input symbols processed at once) based on the analysis in Sec. II. Table V shows that although LUT utilization tends not to depend strongly on the number of simultaneously processed input symbols, block RAM utilization is more effective if a higher level of multi-striding is used. Table V is based on the L7 great set and presents results for the proposed architecture with a shared buffer and pipelined automata for 100 Gbps throughput.

TABLE V
L7 GREAT: RESOURCE UTILIZATION OF 100 GBPS PIPELINED ARCHITECTURE

Input symbols	LUTs	BRAMs
1	17966	69
2	10387	37
4	16570	21

In the case of the 100 Gbps system based on parallel architecture with separate data paths and four input symbols processed at once, it is necessary to distribute packets into $512/8/4 = 16$ data paths.

V. RESULTS

The results are shown in Table VI. Asterisk symbol $(*)$ marks that synthesis for the frequency 200 MHz did not meet timing constraints. In the case of multi-striding, timing constraints are not met because of the critical path in complex next-state logic implemented by a series of LUT elements. Complex next-state logic is caused by states with many input transitions. In the case of parallel architecture with separate data paths, the critical path passes a series of LUT elements that implement transformers of data path width as described in Sec. III.

The graph in Fig. 8 compares LUT utilization of the L7 selected set. We were able to generate a multi-striding circuit only for 10 Gbps and 40 Gbps throughputs.

The graph in Fig. 9 compares LUT utilization of the Snort backdoor set. In this case, we were able to generate a multi-striding circuit only for 10 Gbps throughput.

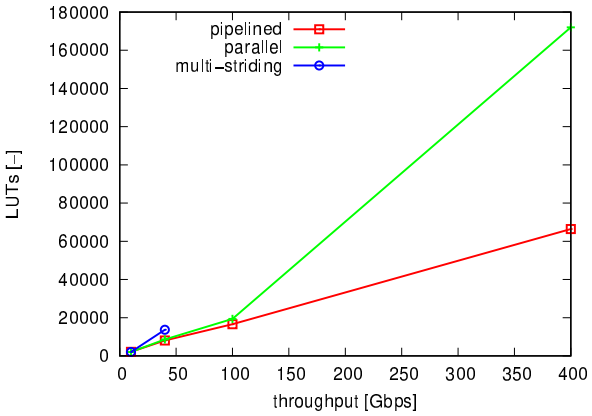

Fig. 8. LUT occupation vs throughput for L7 great

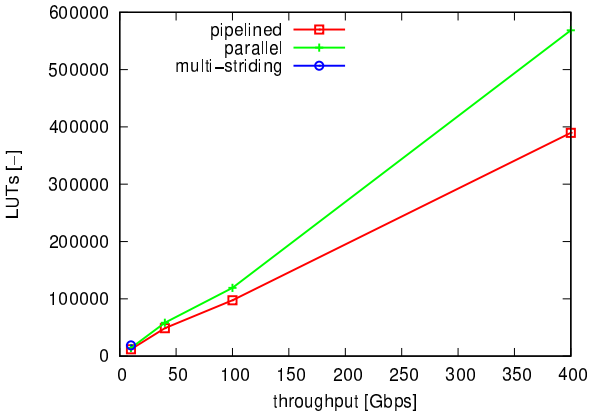

Fig. 9. LUT occupation vs throughput for Snort backdoor rules

The graph in Fig. 10 shows block RAM utilization for parallel architecture with separate data paths and for the proposed architecture with a shared packet buffer and pipelined automata. There is only one graph as block RAM utilization is independent of the set of regular expressions. Multi-striding is not mentioned in the graph in Fig. 10 because it does not use block RAMs. We can see that block RAM utilization grows linearly with the throughput in cases of the proposed architecture while it grows exponentially in cases of parallel architecture.

VI. CONCLUSION

We have shown limited scalability of multi-striding, which is a widely used technique to increase the processing speed of RE matching architectures. Multi-striding causes exponential growth of transition table. We were not able to use multi-striding to achieve 100 Gbps throughput, even for simple NFA-based architecture. The number of FPGA LUT elements grows exponentially and frequency decreases significantly.

Therefore, we have introduced a new scalable architecture for RE matching with 100 Gbps throughput. The architecture uses one input packet buffer and multiple automata in the pipelined structure. The architecture is scalable. The through-

TABLE VI
RESOURCE UTILIZATION

		multi-striding			parallel data paths			pipelined automata		
throughput	data set	LUT	FF	BRAM	LUT	FF	BRAM	LUT	FF	BRAM
10 Gbps	L7 great	2013	289	0	2002	508	2	2118	747	7
	L7 selected	749	116	0	1316	314	2	1482	651	7
	Snort	18721	6265	0	14265	9812	2	11847	7810	16
40 Gbps	L7 great	*13678	*2390	*0	*8614	*2048	*32	8077	2790	13
	L7 selected	N/A	N/A	N/A	*5537	*1272	*32	5582	2393	13
	Snort	N/A	N/A	N/A	*58248	*39264	*32	48818	31914	22
100 Gbps	L7 great	N/A	N/A	N/A	*19388	*4112	*120	16570	5557	21
	L7 selected	N/A	N/A	N/A	*13808	*2560	*120	11100	4705	21
	Snort	N/A	N/A	N/A	*119222	*78544	*120	97460	64042	30
400 Gbps	L7 great	N/A	N/A	N/A	*172009	*149642	*928	66402	21883	69
	L7 selected	N/A	N/A	N/A	*151561	*143434	*928	43489	18439	69
	Snort	N/A	N/A	N/A	*568553	*447370	*928	389758	256676	78

* Timing constraints for 200 MHz not met.

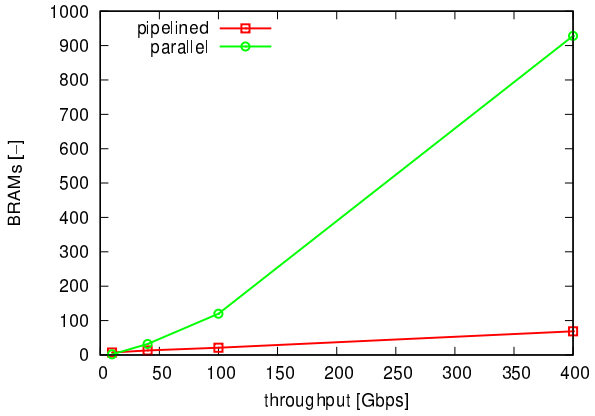

Fig. 10. Block RAM occupation vs throughput (independent of the set of regular expressions)

put can be finely tuned by the number of automata in the pipeline and utilization of FPGA resources grows only linearly with the processing speed, which is a significant improvement to multi-striding.

Moreover, the frequency remains the same despite the increasing number of automata. The architecture uses a direct connection of automata to the input packet buffer and, rather than multiplexing data to automata, it circulates the current state in the pipeline to meet with corresponding input data words. In the case of the 100 Gbps system, this efficient structure of input packet buffer reduces the number of BlockRAMs to 18% when compared to a simple parallel implementation.

The proposed processing can be used for any NFA or DFA architecture where a constant amount of data is processed within one clock cycle. As for future research, we are going to extend the concept to support Delayed input DFA and other architectures, where the input can be stopped for several clock cycles.

ACKNOWLEDGMENT

This paper is based upon work supported by the TACR grant DCPro no. TH01010229, project LM2015042 funded by the Ministry of Education, Youth and Sports of the Czech Republic and Brno University of Technology grant no. FIT-S-14-2297.

REFERENCES

[1] M. Becchi, C. Wiseman, and P. Crowley, "Evaluating regular expression matching engines on network and general purpose processors," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '09. New York, NY, USA: ACM, 2009, pp. 30–39. [Online]. Available: http://doi.acm.org/10.1145/1882486.1882495

[2] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using fpgas," in *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, ser. FCCM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 227–238. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2001.22

[3] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns," in *In Proceedings of 13th International Conference on Field Program*, 2003, pp. 956–959.

[4] I. Sourdis, J. Bispo, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching in reconfigurable hardware," *Journal of Signal Processing Systems*, vol. 51, no. 1, pp. 99–121, 2008. [Online]. Available: http://dx.doi.org/10.1007/s11265-007-0131-0

[5] S. Yun and K. Lee, "Optimization of regular expression pattern matching circuit using at-most two-hot encoding on fpga," in *2010 International Conference on Field Programmable Logic and Applications*, Aug 2010, pp. 40–43.

[6] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '06. New York, NY, USA: ACM, 2006, pp. 339–350. [Online]. Available: http://doi.acm.org/10.1145/1159913.1159952

[7] M. Becchi and P. Crowley, "A-dfa: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 4:1–4:26, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2445572.2445576

[8] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS '06. New York, NY, USA: ACM, 2006, pp. 93–102. [Online]. Available: http://doi.acm.org/10.1145/1185347.1185360

[9] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, April 2004, pp. 249–257.

[10] C. Lin, C. Huang, C. Jiang, and S. Chang, "Optimization of pattern matching circuits for regular expression on FPGA," *IEEE Trans. VLSI Syst.*, vol. 15, no. 12, pp. 1303–1310, 2007. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2007.909801

[11] B. Caswell, J. C. Foster, R. Russell, J. Beale, and J. Posluns, *Snort 2.0 Intrusion Detection*. Syngress Publishing, 2003.

[12] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, no. 23-24, pp. 2435–2463, Dec. 1999. [Online]. Available: http://dx.doi.org/10.1016/S1389-1286(99)00112-7

[13] Y. H. Yang and V. Prasanna, "High-performance and compact architecture for regular expression matching on fpga," *IEEE Transactions on Computers*, vol. 61, no. 7, pp. 1013–1025, July 2012.

[14] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on fpga," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '08. New York, NY, USA: ACM, 2008, pp. 30–39. [Online]. Available: http://doi.acm.org/10.1145/1477942.1477948

[15] Y. Kaneta, S. Yoshizawa, S. i. Minato, H. Arimura, and Y. Miyanaga, "Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching," in *Field-Programmable Technology (FPT), 2010 International Conference on*, Dec 2010, pp. 21–28.

[16] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, May 2007, pp. 1064–1072.

[17] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ser. ANCS '07. New York, NY, USA: ACM, 2007, pp. 155–164. [Online]. Available: http://doi.acm.org/10.1145/1323548.1323574

[18] J. Kořenek, "Fast regular expression matching using fpga," *Information Sciences and Technologies Bulletin of the ACM Slovakia*, vol. 2, no. 2, pp. 103–111, 2010.

[19] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of the 2007 ACM CoNEXT Conference*, ser. CoNEXT '07. New York, NY, USA: ACM, 2007, pp. 1:1–1:12. [Online]. Available: http://doi.acm.org/10.1145/1364654.1364656

[20] ——, "Extending finite automata to efficiently match perl-compatible regular expressions," in *Proceedings of the 2008 ACM CoNEXT Conference*, ser. CoNEXT '08. New York, NY, USA: ACM, 2008, pp. 25:1–25:12. [Online]. Available: http://doi.acm.org/10.1145/1544012.1544037

[21] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 191–202.

[22] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '08. New York, NY, USA: ACM, 2008, pp. 50–59. [Online]. Available: http://doi.acm.org/10.1145/1477942.1477950

A.6 Paper VI

Software Defined Monitoring of Application Protocols

Software Defined Monitoring of Application Protocols

Lukáš Kekely, Viktor Puš
CESNET a. l. e.
Zikova 4, 160 00 Prague, Czech Republic
Email: kekely,pus@cesnet.cz

Jan Kořenek
IT4Innovations Centre of Excellence
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, 612 66 Brno, Czech Republic
Email: korenek@fit.vutbr.cz

***Abstract*—Current high-speed network monitoring systems focus more and more on the data from the application layers. Flow data is usually enriched by the information from HTTP, DNS and other protocols. The increasing speed of the network links, together with the time consuming application protocol parsing, require a new way of hardware acceleration. Therefore we propose a new concept of hardware acceleration for flexible flow-based application level monitoring which we call Software Defined Monitoring (SDM). The concept relies on smart monitoring tasks implemented in the software in conjunction with a configurable hardware accelerator. The hardware accelerator is an application-specific processor tailored to stateful flow processing. The monitoring tasks reside in the software and can easily control the level of detail retained by the hardware for each flow. This way the measurement of bulk/uninteresting traffic is offloaded to the hardware while the advanced monitoring over the interesting traffic is performed in the software. The proposed concept allows one to create flexible monitoring systems capable of deep packet inspection at high throughput. Our pilot implementation in FPGA is able to perform a 100 Gb/s flow traffic measurement augmented by a selected application-level protocol parsing.**

I. Introduction

The task of network traffic monitoring is one of the key concepts in modern network engineering and security. A golden standard in the network monitoring is the basic NetFlow measurement. In NetFlow, the monitoring device collects basic statistics about the IP flows and reports them to a central storage collector in the Cisco NetFlow v5 protocol. NetFlow measurement is a stateful process, because for each packet the flow state record is updated in the device (e.g. counters are incremented), and only the resulting numbers are exported. This also implies that some information is lost in the monitoring process and that the flow collector (where further data processing is usually done) has a limited view on the network. The ability to analyze the application layer in the monitoring process is, therefore, very important in order to improve the quality and flexibility of network monitoring.

The evolution of the NetFlow protocol led to the IPFIX protocol [1]. IPFIX allows for the extension of the exported flow record for any other additional information. While IPFIX solves the task of *transmitting* the additional data, there remains the issue of *obtaining* the additional data. This process inevitably requires additional computational resources.

Pure software implementation of the application level flow monitoring is certainly possible, yet its throughput is limited mainly by the performance of commodity processors. It should be noted that every new packet is inevitably a cache miss in the CPU. Pure hardware implementation, on the other hand, has poor flexibility because the complex protocol parsers are very hard to implement in Hardware Description Languages. Moreover, the evolving nature of network threats and security issues implies the need for a fast change of the monitoring process, which is much more difficult for the hardware. These thoughts lead us to the idea of a hardware accelerator tightly coupled to a software controller with monitoring applications as software plugins.

We focus on the process of obtaining the high-quality, unsampled flow measurement data augmented by application-layer information. Our key idea is that even the advanced application-layer processing usually needs to observe only some flows containing only a small fraction of traffic (such as DNS, with typically no more than 1 % of all packets), or even only a small amount of packets within each of these flows (such as HTTP, typically carrying the HTTP header in the first few packets after the TCP handshake).

We employ a hardware accelerator to perform the offload of the flow measurement for the bulk traffic that is not (or no longer) interesting to the application-layer processing tasks. Also, the hardware accelerator partially has the role of the basic NIC - network interface card. Therefore, it passes a small fraction of the packets intact to the monitoring software and performs flow measurement of the rest.

The use of measurement offload can be easily controlled on a per flow basis by the monitoring software and adjusted to its current needs. Offload control is realized through unified interface by dynamically specifying a set of rules. These rules are then installed into the hardware accelerator to determine interestingness of individual network flows for advanced software processing. Thanks to this unified control interface the proposed system is very flexible and can be used for a wide range of different network monitoring applications. The whole system is designed to be easily extensible by monitoring plugins at the software side. Each monitoring application (in the form of SDM plugin) has three conceptual interfaces: input packets, output measured values, and the control interface to express interest and disinterest in particular fractions of the network traffic. We demonstrate the SDM system on four different monitoring applications: NetFlow measurement, HTTP parsing, a combination of both and DNS protocol parsing.

978-1-4799-3360-0/14/$31.00 ©2014 IEEE

The contribution of our work is three-fold:

- Design of a new concept of extensible high speed network monitoring system. This includes a design of a new application-specific processor for the stateful flow measurement and its controller software. (Chapter II)
- Analysis of network traffic to show the possibilities for the hardware acceleration. Assessment of the system feasibility is based on the analysis. (Chapter III)
- Implementation and evaluation of the system in several use cases. (Chapter IV)

II. System Design

A standard model of the flow measurement widely used in 10 Gbps networks relies on a hardware network card performing a packet capture, sometimes enhanced by a packet distribution among several CPU cores. The captured traffic is then sent over the host bus to the memory, where packets are processed by the CPU cores. This model cannot be applied to 100 Gbps networks due to two major performance bottlenecks. First, the throughput of today's PCI Express busses is insufficient. The second bottleneck lies in limited computational power which is insufficient for advanced monitoring tasks. We propose a new acceleration model which overcomes the above-mentioned bottlenecks by a well-defined hardware/software co-design. The main idea is to give the hardware the ability to handle basic traffic processing. Only a granular control of the HW and some more advanced tasks are left for the software.

The basic idea of acceleration by the SDM system is based on a finely controlled data loss and data distribution realized by hardware preprocessing of the network traffic. The preprocessing is fully controlled by the software applications. Therefore, the first few packets of a new flow are sent to the software, which decides which type of hardware preprocessing will be used for the following packets of the flow. There are two basic options for the hardware acceleration:

- It is possible to extract the interesting data from packets in the hardware and send them only to the software in a predefined format, which we call a Unified Header (UH). Then only a few bytes for each packet are transferred through the PCI Express bus and the CPU has a lower load too because the packet parsing is done in the hardware.
- Furthermore, packets can be aggregated to NetFlow records directly in the HW which brings even higher performance savings.

Some advanced monitoring applications perform deep packet inspection on interesting fragments of traffic and, therefore, have to analyze the whole packets. For example, extraction of information from HTTP headers needs several first packets for each HTTP flow. Therefore, the proposed system provides a control over the hardware packet preprocessing at the flow level granularity.

The top-level conceptual scheme of the proposed SDM system is shown in Fig. 1. Data paths are represented by black arrows and control paths by red arrows. The system is composed of two main parts (firmware and software) connected together through the PCI Express bus. The processing of all incoming packets starts with the header parsing and extraction of interesting metadata (Header Field Extractor - HFE block). Extracted metadata are then used to classify the packet based on a software defined set of rules (Classifier block). Each rule identifies one specific flow and defines a method of hardware preprocessing of its packets. More precisely, each rule specifies the type of packet preprocessing and the target software channel. Packets can be processed in a hardware flow cache, dropped, trimmed or sent to the software unchanged or in the form of a Unified Header (UH Generator block). Flow records in the hardware flow cache are periodically exported to the software. Sending the data to the software is realized by the direct memory accesses (DMA) over the PCI Express bus. There are multiple independent logical DMA channels with the corresponding DMA buffers in the host RAM to aid parallel processing by a multicore CPU.

The data can be stored in DMA buffers in the form of whole packets, Unified Headers or flow records. This data can be monitored by the set of user specific software applications such as the flow exporter which analyzes the received data and exports the flow records to the collector. User applications can read the data from the selected DMA channels and can also specify which types of traffic they want to inspect and which flows can be preprocessed in hardware. For example, an HTTP header parser needs to inspect every packet in the HTTP flow until it acquires the required information (e.g. the URL). Definitions of interesting and uninteresting bulk traffic from all applications are passed to the SDM controller. The SDM controller aggregates the definitions into rules and configures the firmware behavior in order to achieve the maximal possible reduction of the traffic resulting in maximal hardware acceleration.

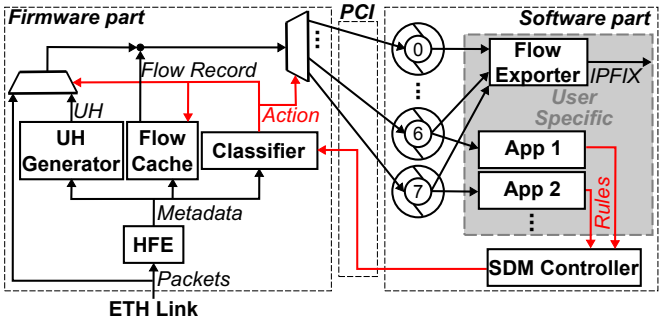

Fig. 1. Conceptual top-level scheme of SDM system

A different view of the proposed SDM system is shown as a layered scheme in Fig. 2. The SDM system is designed to work on a hardware accelerated network board with an FPGA chip. Our implementation uses a custom made board with 100 Gb/s Ethernet interface and Virtex-7 FPGA with the NetCOPE platform [2] realizing the basic network traffic capture and communication with the software (DMA). The core of the FPGA firmware is realized by the firmware part of the SDM system described earlier, which is able to process the incoming traffic at full speed of the network link. The software layer of the SDM includes means for the basic configuration of the firmware, network data transfer (black Data Path) and control of SDM firmware (red Control Path). Data can be received from the firmware in the standard PCAP or the proprietary SZE

format. On the top of the SDM system, there are individual user specific software applications.

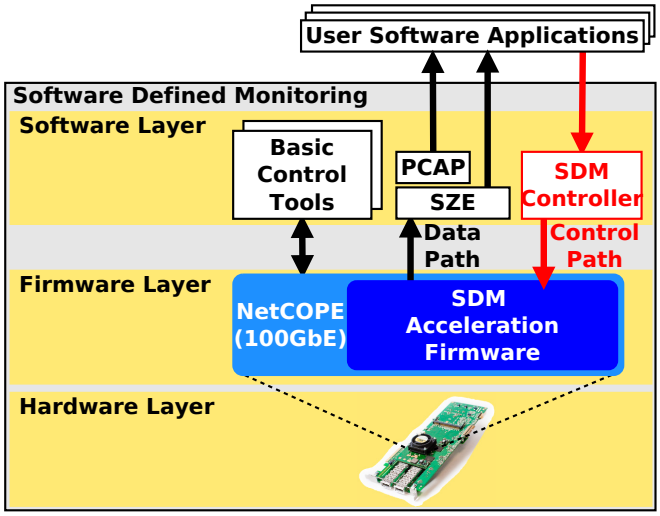

Fig. 2. Layered model of SDM acceleration system

Fig. 3 shows a top level implementation scheme of the SDM firmware. The main firmware functionality is realized by the processing pipeline of four modules: Header Field Extractor (HFE), Search, Update and Export. This pipeline processes the incoming network traffic and creates an outgoing data flow for the software. Incoming frames do not flow directly through the processing pipeline, but are rather stored in a parallel FIFO. The processing pipeline uses only metainformation extracted from frames headers (UH). Whole software control of the processing pipeline is managed by the SW Access module which configures preprocessing rules used in the Search unit. In order to achieve sufficient capacity for rules and flow records, the firmware stores them in external memory (Table1 and Table2). Access to the external memory is managed by Memory Arbiter.

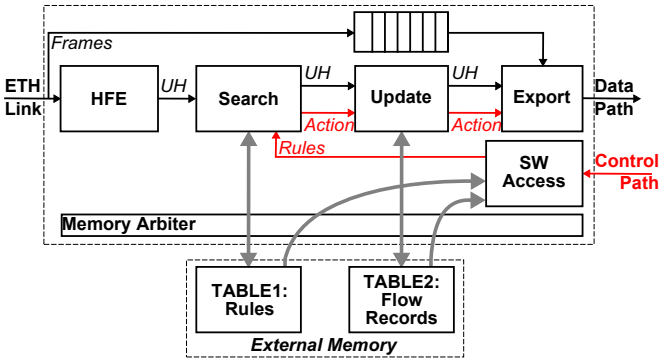

Fig. 3. Detailed firmware scheme

As already described, the SDM firmware functionality is realized by 6 main modules:

- **Header Field Extractor** analyzes headers of incoming frames and extracts interesting information from them, especially fields that clearly identify network flows. In order to identify flows we use the classical 5-tuple: source and destination IP addresses, source and destination TCP/UDP port numbers and a protocol number. We use our own flexible low latency modular implementation of the header parser [3].
- **Search** assigns an action to every processed frame based on its flow identifier. An action assignment is realized using a set of software defined rules in the form of a flow identifier paired with action (Table1 in external memory). Management of the rule set is possible through a control interface capable of an atomic add, remove or update of the rules. A frame classification by the Search unit works in 2 steps. Firstly, the frames are assigned with an action based on a small set of relatively static rules on flow groups (e.g. flows with source port 80). Secondly, the action from the first step can be further particularized by a set of dynamic rules for individual flows. Standardly, user applications set up rules of the first type during startup and then they manage the set of second type rules during traffic processing.
- **Update** manages the records for flows in Table2. It mainly actualizes their values based on input UH and its action. The action for every UH has the address of the record and a specification of the operation (aggregation type). Update of the record is realized by two memory operations: read actual values of the record fields and write back the updated values. Another operation is the export of the record values, possibly followed by the reset of the record values in the memory. Records can be exported not only at the flow end but also in a periodical manner, so that the software applications can have actual information about hardware monitored flows. Control of memory allocation for records and their periodical export is realized by SDM control software.
  In the first version of SDM we implement only the simple NetFlow aggregation as the record update operation – increase packets/bytes counters, update flow start/end timestamp and logical or of TCP flags. It is however possible to support more types of records and operations in the future.
- **Export** pairs together corresponding UH transaction with frame data from FIFO memory. Then it chooses the DMA channel and format for the data based on action assigned by the Search module.
- **SW Access** is the main access point into the SDM firmware from the software. Its primary function is to manage the rules and to initiate the export of the flow records based on software commands. Besides, it contains all state and control registers. It also enables direct software access into external memory (still used only for debug).
- **Memory Arbiter** provides and manages access to the external memory. Its main responsibilities are proper interleaving of memory accesses and routing of read data between units. It also ensures atomicity and deterministic succession of all memory operations.

The network traffic preprocessing by firmware is controlled from the software. The core of the controlling software are the

monitoring applications. Each monitoring application has the form of an SDM plugin. The main input to the plugin is the data path carrying the packets, UHs or flow records. The plugin output is the data that the plugin has parsed/detected/measured. This output data is then added to the exported IPFIX flow record. The third interface of the monitoring application is the interface to the SDM Controller.

From the application view, the SDM controller accepts the preprocessing requests from multiple applications, aggregates them and administers them into the firmware. In order to achieve that, the controller performs the following operations:

- On the fly management of the set of applications currently controlling the firmware preprocessing.
- Preprocessing requests reception from applications.
- Storing and aggregation of the received preprocessing rules (requests).
- Timed expiration of application rules.

The aggregation of preprocessing rules is based on different degrees of data reduction. Ordered from the lowest degree of data reduction the preprocessing types are: none (whole packets), partial (UH), complete (flow record) and elimination (packet drops). Therefore, aggregation of rules in the SDM controller is done simply by the selection of the lowest preprocessing degree (highest data preservation) for particular flows which satisfy the information level requirements of all applications.

When configuring the firmware, the SDM controller communicates directly with the SW Access module. In order to maintain a proper functionality of SDM firmware, the controller must carry out the following operations:

- Management of rules activated in the firmware (rule add/delete/update) based on the application demands.
- Cyclic export of active flow records computed in the firmware flow cache.
- Allocation of records in the firmware flow cache.

III. Proof of Concept

This chapter analyzes the proposed concept. It is divided into three sections. The first section proposes several possibly weak points of the SDM concept. The second section presents an analysis of network traffic. The aim of the analysis is to show whether the SDM concept is a sound idea. The third section draws conclusions about the presented analysis and addresses all of the proposed weak points.

A. Potential Weak Points

From the presented SDM concept one can infer several potential weak spots in the system design. Their existence can (in bad circumstances) lead to lower effectiveness of hardware preprocessing usage and therefore to a low degree of achieved application acceleration. Major recognized potential weaknesses of the SDM design are the following:

- **Long duration of the feedback loop.** In order to maintain a throughput of 100 Gbps and more, the hardware processing of packets cannot wait for software decisions – the packets must be processed on the fly. Therefore, the action chosen for the flow does not affect a certain amount of leading packets from this flow. If a high portion of flows on the monitored link have an extremely short duration, the acceleration ratio achievable from the usage of SDM declines.
- **Limited firmware capacity.** Because of the fine granularity of preprocessing control, the firmware must store some information about each known flow. The capacity of table with search rules or flow records in the firmware (Table1 or Table2 in Fig. 3) can be restrictive. An extremely high number of concurrent flows on the network can restrict the preprocessing usage to only a small portion of the flows. Negative effects of this restriction can be significantly reduced by an adequate selection of preprocessed flows. Suitability of the flow is given by the achievable reduction of its data during preprocessing. It is generally desirable to prefer the preprocessing of large (heavy) flows.
- **Insufficient data reduction.** Hardware preprocessing reduces the data quantity from the network by converting the packets into Unified Headers, aggregating them into flow records or by dropping them completely. The amount of data reduction is directly proportional to the size of processed packets and flows. Therefore, in the case of extremely short flows with very short packets the effectiveness of data reduction of the SDM can be relatively small.
- **Overly granular control.** The choice of the acceleration control basic unit affects the number of required rules in the Search module and the rate of their creation. The benefit from a preprocessing rule covering a small portion of the incoming traffic is small. In the extreme case, the overhead of rule creation can even outweigh the SDM benefits. Also, rule generation in case of extremely small units of control can exceed the achievable throughput of the configuration interface.

B. Network Traffic Analysis

The magnitude of possible negative impacts of the described weak spots is closely related to the character of processed data. Therefore, we have analyzed the properties of the network traffic in a real high-speed backbone network. Based on the measured characteristics we have proven that the proposed SDM system can perform very well when deployed in real networks.

All of the measurements in this paper were conducted in the high-speed CESNET2 backbone network. CESNET2 is Czech NREN which has optical links operating at speeds up to 100 Gbps and routes mainly IP traffic. We conducted all of our measurements during the standard working hours of the workweek. We measured mean size of packets in bytes, mean size of flows in packets and mean time duration of flows. Because we aim for the application protocols, we measured the mentioned characteristics, not only for the whole network traffic on the link, but also for the selected application protocols. We selected a set of interesting protocols: HTTP, HTTPS, DNS, SMTP, SSH and SIP. Furthermore, we measured

the percentage of these protocols in the captured traffic in the matter of flows, packets and bytes.

The results of the basic network traffic analysis are shown in Table I. The table shows that the statistics vary depending on the application protocol. Dominant is the HTTP protocol with more than a quarter of all flows and more than a half of all packets and bytes. Moreover, HTTP flows and packets are generally larger (heavier) and longer. A considerable amount of traffic belongs to HTTPS, which has generally smaller and longer flows than HTTP. A high amount of flows also belong to the DNS protocol (one fifth), but this number is highly disproportional to the DNS total packet and bytes percentage. DNS flows are generally very small (light). A majority of them consists of only one small packet.

	Flows [%]	Packets [%]	Bytes [%]	Flow [packets]	Flow [s]	Packet [Bytes]
HTTP	25.45	54.36	58.68	63.1	7.167	963.2
HTTPS	14.28	6.92	4.75	14.3	8.493	611.7
DNS	18.89	0.72	0.17	1.1	0.179	207.2
SMTP	0.38	0.22	0.14	17.2	2.934	573.8
SSH	0.04	0.01	0.00	11.6	17.433	233.0
SIP	0.00	0.00	0.00	4.9	24.701	420.9
others	40.96	37.76	36.26	27.3	7.735	856.7
all				29.6	6.257	892.2

TABLE I. BASIC STATISTICAL CHARACTERISTICS OF NETWORK DATA GROUPED BY THE APPLICATION PROTOCOL

Another interesting characteristic of the network is the *distribution* of packet lengths. The majority of packets are either very long (over 1300 B: 57 %) or very short (under 100 B: 35 %). Especially dominant are both extremes from the range of lengths supported by the Ethernet standard – 42 and 1500 B. Medium sized packets are not very common.

There is already information about mean flow durations for the selected application protocols in Table I. Further information about the flow time durations can be seen in Fig. 4. Each line in the graph shows the percentage of flows that last shorter than the given duration. Generally (red thicker line) over $\frac{2}{3}$ of all flows are shorter than 100 ms and only a tenth of them exceed a duration of 10 s. Also majority of DNS and SIP flows have a duration under 10 ms.

Fig. 4 shows further information about flow duration, but does not say anything about time distribution of packets inside the flows. Weights of individual flows are also not considered. A better look at packet timing inside the flows can be shown by measuring the relative arrival times of packets from the start of the flow. Thus, the first packet of each flow has the zero relative arrival time and its absolute arrival time marks the starting time of that flow. Then, each consequent packet has a relative arrival time equal to the difference of its absolute arrival time and the marked start of the flow. Results of this measurement are shown in Fig. 5. The graph shows that generally (red thicker line) only a small portion of all packets arrive right after the start of the flow – only a fifth of all packets arrive during the first second of flow. This fact leads to the conclusion that flows with short duration carry only a very few packets. The conclusion is further strengthened by the fact that the majority of flows have a very short duration. There are exceptions such as DNS and SIP though.

There is already information about mean flow sizes for selected application protocols in Table I. Further information

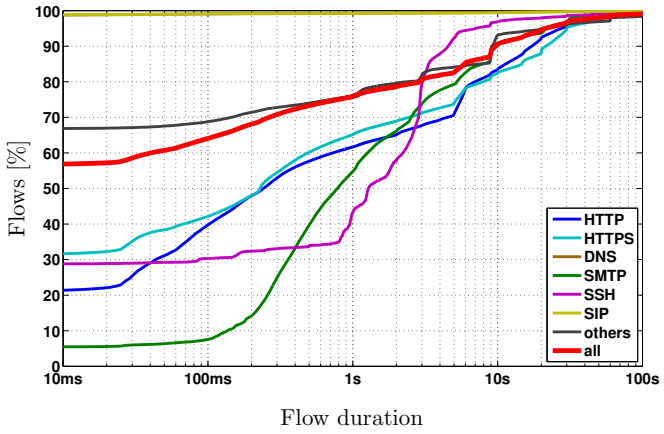

Fig. 4. Cumulative distribution functions of flow durations

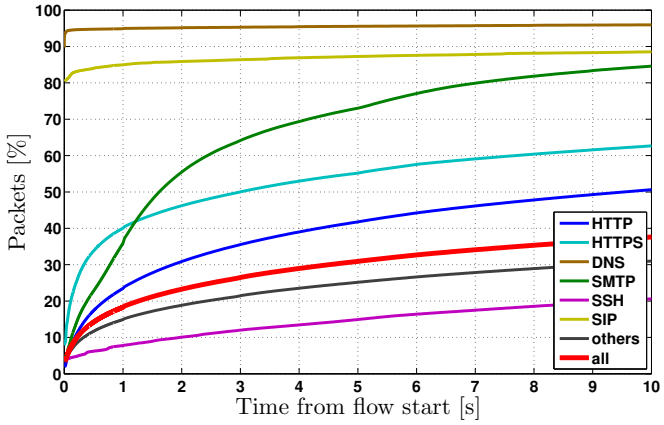

Fig. 5. Cumulative distribution functions of packet arrival times

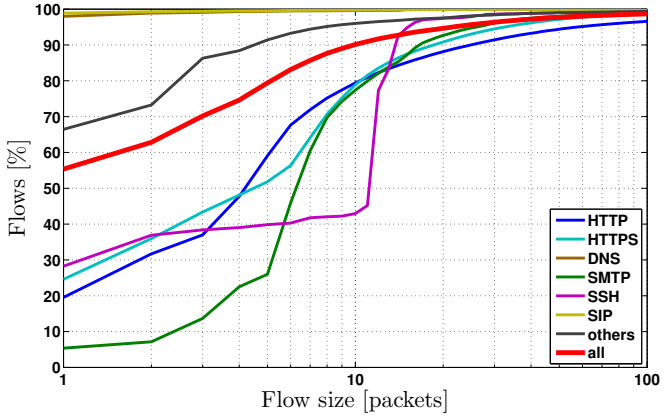

Fig. 6. Cumulative distribution functions of flow sizes

about flow sizes can be seen in Fig. 6. Each line of the graph shows the percentage of flows that consists of less packets than a given number. Generally (red thicker line) only a tenth of all network flows have more than 10 packets. Also, virtually all DNS and SIP flows consist of a single packet.

Fig. 6 shows further information about flow sizes, but does not clearly say anything about the percentage of all packets carried by flows of different sizes. It is known that

high-speed network traffic has a heavy-tailed character of flow size distribution. The heavy-tailed character of flow size distribution derived from the measured values is shown in Fig. 7. The graph shows the portions of all packets carried by the specified percentage of the heaviest flows on the network. It can be seen that generally (red thicker line) 0.1 % of the heaviest flows carries around 60 % of all packets and 1 % carries even around 85 %. An exception to the heavy-tailed distribution of flow sizes is the DNS protocol. On the other hand, SIP and SSH protocols have a heavier tail than average.

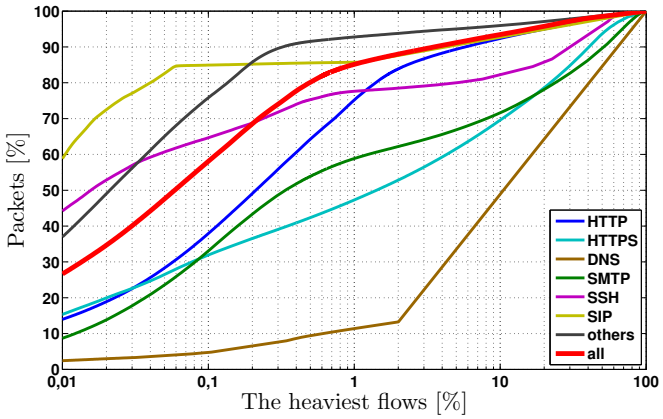

Fig. 7. Portions of packets carried by the percentage of the heaviest flows

A consequence of the heavy-tailed character of the network traffic is that even by selecting a small percentage of the heaviest flows, we can cover the majority of packets. The problem then lies in the effective prediction of which flows are among the heaviest. More accurately, it lies in the capability to recognize the heaviest flows only from the properties of their first few packets. The simplest method of this recognition is based on the rule that every flow is considered heavy after the arrival of its first k packets for some selected decision threshold k. The main advantage of this method is its simplicity – no packet analysis nor advanced stateful information for the flows is needed.

The measured accuracy of the heaviest flow selection by the described simple method is shown in Fig. 8 and Fig. 9. These graphs show the relations between the value of threshold k to the portion of heavy marked flows (first graph) and packets covered by them (second graph). By a combination of values from both graphs we can see that with the rising decision threshold the portion of flows marked heavy dramatically decreases, but the percentage of covered packets decreases rather slowly. For example, decision threshold $k = 20$ leads to only 5 % of heavy marked flows covering around 85 % of all packets. Exceptions are the DNS and to some extent also HTTPS and SMTP protocols, where the percentage of covered packets decreases quickly.

A different view of the simple heavy flow prediction method effectiveness can be seen in Fig. 10. It shows the mean number of packets covered by one heavy marked flow for different values of the decision threshold k. Values shown in the graph rise with the decision threshold to a considerably higher number than the mean sizes of the flows from Table I – hundreds or even thousands of packets instead of only tens

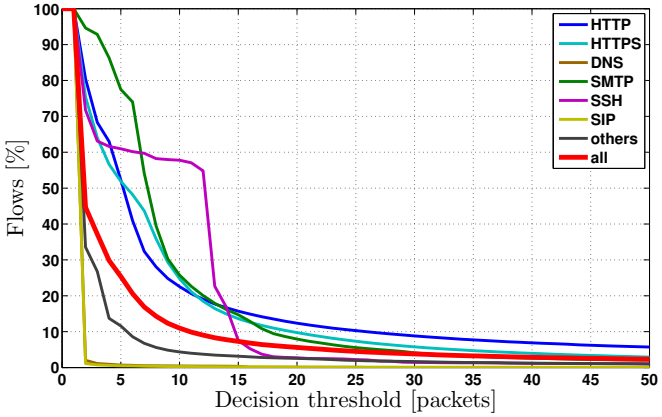

Fig. 8. Heavy flow detection using the simple method – portions of selected flows

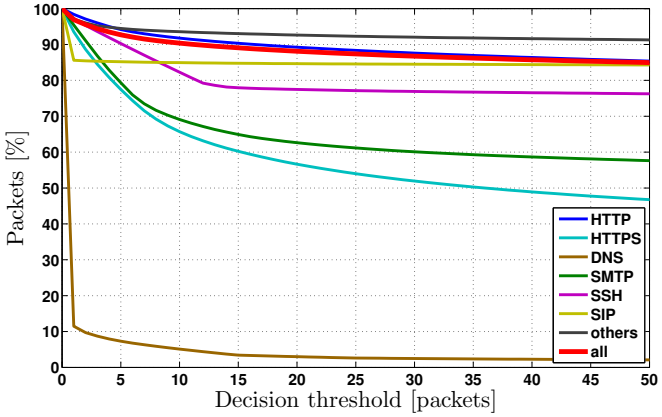

Fig. 9. Heavy flow detection using the simple method – portions of captured packets

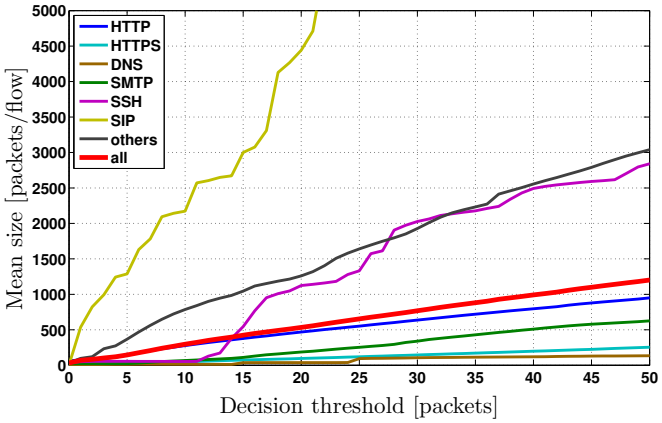

Fig. 10. Mean number of captured packets per flow in flows selected using the simple method

of them. This clearly proves that even a simple heavy flow prediction method effectively predicts the heaviest flows.

C. Proof of Concept Conclusion

Based on the analysis results presented in this section we can now draw conclusions about the negative effects of

possible weak spots of the SDM design. The conclusions are:

- **Long duration of feedback loop.** The expected SDM feedback loop delay is in the area of tens to hundreds of milliseconds. Fig. 4 shows that the majority of flows has a duration too short for this requirement (over $\frac{2}{3}$ shorter than 100 ms). But in spite of that, the majority of packets is carried by longer flows and arrives later from the flow start (only a tenth of packets during the first 100 ms according to Fig. 5). These results lead to a small negative effect of feedback loop duration on the system performance.
- **Limited firmware capacity.** Fig. 7 shows a heavy-tail character of network traffic. Moreover, figures 8 and 9 show that even a very simple heavy flow prediction method can give very good results. In conclusion, even with a relatively small number of flow rules it is possible to cover the majority of packets.
- **Insufficient data reduction.** Unified Headers and Flow Records have sizes of tens of bytes. Table I shows that rather large packets are mostly used – the mean size is nearly 900 B. Therefore, a reduction of network traffic bytes is sufficient.
- **Overly granular control.** Fig. 10 shows that with an appropriate selection of flows it is possible to achieve a high effectiveness of rules. Each rule can specify a preprocessing offload into HW of hundreds or even thousands of packets on average.

From these conclusions it is clear that possible weak spots of the SDM design will not have a large negative impact on system performance in real networks. Exceptions are protocols like DNS with a very high percentage of single packet flows. Fortunately, these protocols cover only a small portion of network traffic (e.g. DNS with less than 1 %).

IV. Results

In order to verify the proposed system further, we have implemented the whole SDM system prototype. The hardware part of the system is realized by the accelerator board with the powerful Virtex-7 H580T FPGA. The whole FPGA firmware occupies less than half of the available FPGA resources. That includes not only the SDM functionality, such as packet header parsing and NetFlow statistics updating, but also 100 Gbps Ethernet, PCI-Express and QDR external memory interface controllers. The software is realized as a set of plugins for the Invea-Tech's Flowmon exporter software [4]. This exporter allows us to modify its functionality to the extent required by the SDM system.

The designed SDM system brings acceleration of monitoring applications based mainly on software defined hardware acceleration of network traffic preprocessing. Control of the preprocessing is mainly realized by the monitoring applications through on the fly defined dynamic rules for particular flows. These rules are generated as a reaction to the first few packets of the flow. Therefore, there is some delay between the flow start and rule application. The duration of this delay influences the portion of packets affected by the rules. The basic view of achievable SDM system effectiveness can be gained from an examination of an achievable portion of packets whose preprocessing was influenced by the dynamic flow rules.

In order to test the described ability of the SDM system we created a simple use case. In this use case, only a specified number of the first packets from each flow is interesting to the software. All packets from unknown (new) flows are, therefore, by default forwarded into the software application. SDM controller software counts the number of packets in each flow. Right after the reception of the specified number of packets for a flow, the application creates a rule for the firmware to drop all the following packets from this flow. This decision method is absolutely the same as the simple heavy flow detection method defined in the previous section.

In the described test case we have measured the portion of packets dropped by the SDM firmware. The results are projected into the graph in Fig. 11. The graph shows the percentage of dropped (influenced) packets (solid lines) and the percentage of flows for which the rule was created (dashed lines). For comparison, analytical results from graphs 8 and 9 in the previous section are also shown (red). The result is that the SDM system can influence preprocessing of up to 85 % of all packets from real network traffic by dynamic flow rules. A visible difference of about 10 % of influenced packets between analytical and real results is caused by neglecting the duration of rule creation and activation process in the analytical result.

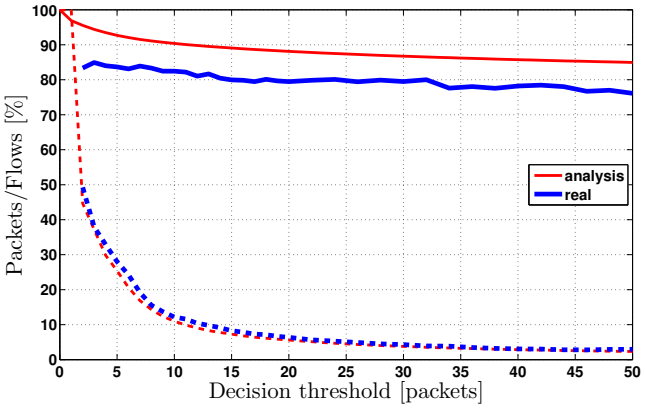

Fig. 11. Portions of offloadable packets and flows using the simple heavy flow detection method

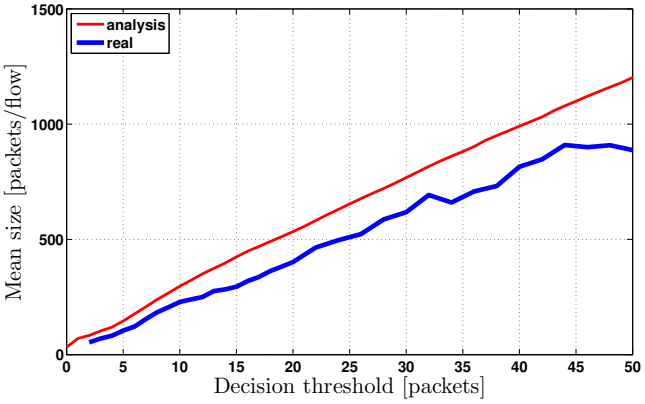

Fig. 12. Mean number of offloadable packets per flow in flows selected using the simple heavy flow detection method

The graph in Fig. 11 also shows a similar character of packets and flows portions as described in the previous section – considerably faster decline in the percentage of flows than in the percentage of packets. A better view is provided in Fig. 12. There, the relation of the mean number of packets influenced by one created rule over the decision threshold value is shown (blue). The red line is analytical result of simple heavy flow detection method effectiveness taken from Fig. 10. The graph shows that real measured effectiveness of this method is slightly worse than the analysis suggests. But it is still very effective and suitable for real usage.

Apart from this artificial use case, we also tested SDM acceleration abilities in more realistic use cases. We tested the performance of the system in the following four cases:

- **Standard NetFlow measurement.** In this use case, all packets from the link are taken into account. By default, they are sent to the software in the form of UH. The software adds dynamic rules to offload the NetFlow measurement of heavy flows (predicted by the simple method) into the hardware accelerator.
- **HTTP header analysis.** We choose HTTP because HTTP traffic is dominant in the networks. Therefore, the acceleration of its analysis is of high importance. In this use case we tested the application that parses HTTP headers and extracts some interesting information (e.g. URL, host, user-agent) from them. Extracted information can then be used to augment the flow records. Because the application works with the data of HTTP packets, only the packets with a source or destination port 80 are sent into the software by default. Others are dropped in the hardware. Furthermore, the application adds dynamic rules to drop the packets of HTTP flows in which it already detected and parsed the HTTP header.
- **Standard NetFlow enriched by HTTP analysis.** This case combines the two previous ones. Both applications are active at the same time without the need of any changes in them. Their traffic requirements are automatically combined by the SDM controller.
- **DNS security analysis.** We choose DNS because it is a bit different from the other protocols. Its flows are extremely short. Therefore, the dynamic flow rules have virtually no effect on DNS preprocessing. But the DNS traffic takes up less than a hundredth of all network traffic. So, even with the use of default rules only (no dynamic rules), SDM should be able to massively accelerate the analysis.

The results of the SDM system testing in the described use cases are shown in Figures 13 and 14. The figures show the portions of all incoming packets and bytes preprocessed in the hardware by a particular method. These hardware preprocessing utilizations lead to a reduction of software application load displayed in Table II. The table shows portions of incoming packets and bytes that are processed by software applications in particular use cases relative to the state without the SDM accelerator. It also shows the percentage of flows for which the rule is created in the hardware.

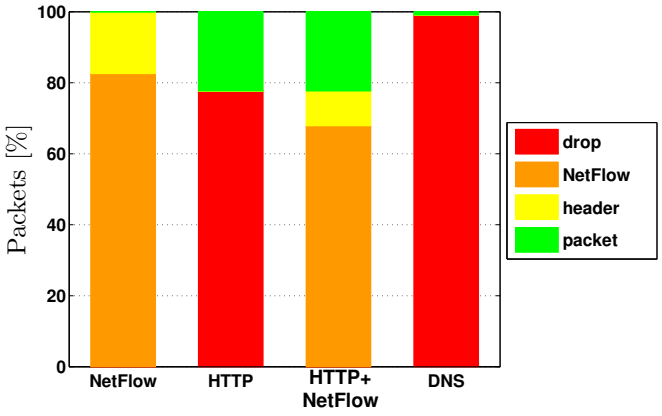

Fig. 13. Portions of hardware preprocessing types in tested use cases

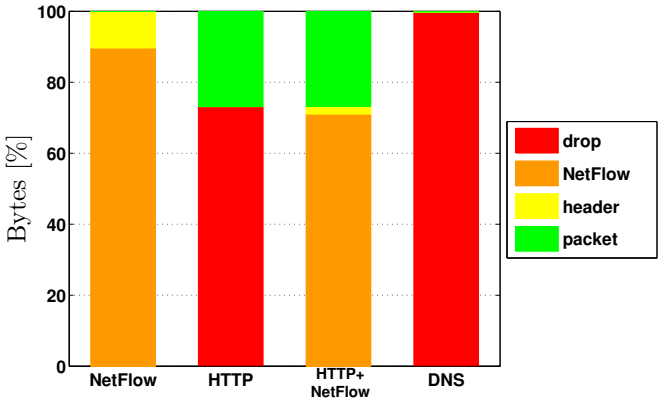

Fig. 14. Portions of hardware preprocessing types in tested use cases

	SW load [%]		Rules in HW
	Packets	Bytes	[% of flows]
NetFlow	17.55	0.86	12.16
HTTP	22.32	26.85	3.60
HTTP+NetFlow	32.42	27.30	11.84
DNS	0.73	0.16	0.00

TABLE II. SOFTWARE APPLICATIONS LOAD USING SDM IN TESTED USE CASES, RELATIVE TO THE STATE WITHOUT THE SDM ACCELERATOR

Standard NetFlow measurement is mostly accelerated by the hardware flow cache. In this way, the software application load is reduced to less than a fifth of all packets (in the form of UH or flow record). Further acceleration rises from the fact that only UHs and flow records are sent to the software, instead of complete packets. The software, therefore, does not parse packets anymore and the PCI Express load is reduced to less than one percent.

SDM accelerates the analysis of application protocols by packet dropping based on static and dynamic rules. This leads to the HTTP parser load being reduced to only about a fifth of all packets and bytes and to the DNS parser load reduced to less than a hundredth.

When the standard NetFlow measurement and the application protocol parsing are used simultaneously, the load of the application protocol parser is the same as when used alone thanks to the DMA channel traffic splitting supported by the SDM. The HTTP parser software still receives only