BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

MAPPING OF PACKET PROCESSING FROM P4 LANGUAGE TO FPGA TECHNOLOGY

MAPOVÁNÍ ZPRACOVÁNÍ PAKETŮ POPSANÉHO V JAZYCE P4 DO TECHNOLOGIE FPGA

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR **Ing. MICHAL KEKELY**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. JAN KOŘENEK, Ph.D.**
ŠKOLITEL

BRNO 2023

Abstract

This thesis deals with the design of novel hardware architectures for packet classification. The main goal is to propose general and flexible hardware approaches capable of classifying packets on high-speed computer networks. The approaches need to be configurable via P4 language description and need to be scaleable to 100 Gbps and faster networks. The thesis starts with an analysis of the current state of the art in packet classification on high-speed networks. Based on the analysis, new architectures for packet classification are proposed. The architectures are designed with scalability, flexibility, and memory efficiency in mind. The goal is to achieve high throughput while maintaining P4-programmability and the ability to carry out general packet classification. Proposed approaches are further optimized and extended to be as efficient as possible. The first architecture uses the DCFL algorithm extended by a parallel TCAM memory, memory duplication and ruleset analysis. The goal is to achieve general packet classification, which has small memory requirements and offer a trade-off between the achieved throughput and the memory requirements. The second proposed approach is more specialized. It optimizes exact match packet classification by leveraging the distributed memories on FPGAs to speed up the Cuckoo hashing algorithm. The main goal is to achieve very high throughputs efficiently. Both approaches are further extended by proposing a caching mechanism that enables efficient external memory usage. Finally, all of the proposed mechanisms are evaluated on real network data, and the achieved results are shown.

Abstrakt

Táto dizertačná práca sa zaoberá návrhom nových hardvérových architektúr na klasifikáciu paketov. Hlavným cieľom je navrhnúť všeobecné a flexibilné hardvérové prístupy, ktoré sú schopné klasifikovať pakety na vysokorýchlostných počítačových sieťach. Prístupy musia byť konfigurovateľné pomocou popisu v jazyku P4 a musia byť škálovateľné na siete s rýchlosťou 100 Gb/s a viac. Práca začína analýzou aktuálneho stavu poznania v oblasti klasifikácie paketov. Na základe tejto analýzy sú navrhnuté nové architektúry pre klasifikáciu paketov. Pri návrhu sa dbá na škálovateľnost, flexibilitu a pamäťovú efektivitu. Cieľom je dosiahnuť vysokú priepustnosť a zároveň udržať programovateľnosť pomocou P4 a schopnosť vykonať všeobecnú klasifikáciu paketov. Navrhnuté prístupy su optimalizované a rozšírené, aby boli čo najefektívnejšie. Prvá architektúra využíva algoritmus DCFL rozšírený o paralelnú pamäť typu TCAM, duplikáciu pamätí a analýzu množiny pravidiel. Cieľom je dosiahnutie všeobecnej klasifikácie paketov, ktorá má nízke pamäťové nároky a ponúka možnosť škálovať priepustnosť za cenu zvýšených zdrojov. Druhý navrhnutý prístup je špecializovanejší. Optimalizuje klasifikáciu paketov založenú na úplnej zhode. Toto je dosiahnuté využitím distribuovaných pamätí na čipe FPGA na zrýchlenie algoritmu kukučieho hešovania. Hlavným cieľom je dosiahnuť veľmi vysokú priepustnosť efektívne. Architektúry sú ďalej rozšírené navrhnutím mechanizmu vyrovnávacej pamäte, ktorá dovoľuje efektívne použiť externé pamäťové bloky. Nakoniec sú tieto architektúry vyhodnotené na skutočných sieťových dátach a sú ukázané dosiahnuté výsledky.

Keywords

P4, classification, hash table, cuckoo hashing, trie, FPGA, packet filtering, DCFL, cache, optimization, high-speed networks

Klíčová slova

P4, klasifikácia, vyhľadávacia tabuľka, kukučie hešovanie, trie, FPGA, filtrovanie paketov, DCFL, vyrovnávacia pamäť, optimalizácie, vysokorýchlostné siete

Reference

KEKELY, Michal. *Mapping of packet processing from P4 Language to FPGA Technology.* Brno, 2023. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Jan Kořenek, Ph.D.

Rozšírený abstrakt

Táto dizertačná práca sa zaoberá návrhom nových hardvérových architektúr na klasifikáciu paketov. Hlavným cieľom je navrhnúť všeobecné a flexibilné hadrvérové prístupy, ktoré sú schopné klasifikovať pakety na vysokorýchlostných počítačových sieťach. Prístupy musia byť konfigurovateľné pomocou popisu v jazyku P4 a musia byť škálovateľné na siete s rýchlosťou 100 Gb/s a viac.

Práca začína analýzou aktuálneho stavu poznania v oblasti klasifikácie paketov. V dnešnej dobe dochádza k neustálemu zrýchľovaniu sieťových liniek a zároveň sa samotné počítačové siete stávajú komplexnejšími a dynamickejšími. Tieto trendy reflektuje aj softvérovo definované sieťovanie (SDN) a jazyk P4. Oba prístupy poskytujú flexibilnejšie a abstraktnejšie spôsoby popisu sieťovej funkcionality. Túto flexibilitu je však potrebné skĺbiť s narastajúcimi rýchlosťami. Existujúce prístupy ku klasifikácii paketov sú často optimalizované na konkrétne prípady použití a neposkytujú možnosť škálovania a rozšírenia. Ďalším nedostatkom býva neefektívna hardvérová implementácia, ktorá znemožňuje využitie technológie FPGA, ktorá poskytuje výkon potrebný na spracovanie paketov na vysokorýchlostných sieťach. Na základe analýzy je zostavená základná pracovná hypotéza, ktorá hovorí, že aj napriek týmto nedostatkom je možné, využitím nových hardvérových architektúr a jazyka P4, dosiahnuť vysokú priepustnosť (100 Gb/s, 200 Gb/s, 400 Gb/s) klasifikácie paketov a zároveň zachovať potrebnú flexibilitu.

V práci sú navrhnuté nové architektúry pre klasifikáciu paketov. Pri návrhu sa dbá na škálovateľnosť, flexibilitu a pamäťovú efektivitu. Cieľom je dosiahnuť vysokú priepustnosť a zároveň udržať programovateľnosť pomocou P4 a schopnosť vykonať všeobecnú klasifikáciu paketov. Navrhnuté prístupy sú optimalizované a rozšírené, aby boli čo najefektívnejšie. Prvá architektúra využíva algoritmus DCFL rozšírený o paralelnú pamäť typu TCAM. Cieľom je dosiahnutie všeobecnej klasifikácie paketov, ktorá má nízke pamäťové nároky a ponúka možnosť škálovať priepustnosť za cenu zvýšených zdrojov. Škálovateľnosť je dosiahnutá pomocou duplikácie pamätí a techník na zníženie pamäťovej náročnosti riešenia. Architektúra je navyše optimalizovaná analýzou množiny klasifikačných pravidiel, na základe ktorej je možné odstrániť pravidlá, ktoré vytvárajú v algoritme DCFL úzke hrdlo. Tieto pravidlá sú klasifikované samostatne v paralelnej pamäti typu TCAM. Výsledná architektúra je schopná dosiahnuť priepustnosť 100 Gb/s a je škálovateľná. Napríklad, na dosiahnutie priepustnosti 10 Gb/s potrebuje pre 5500 pravidiel architektúra iba 20 pamätí BRAM. Architektúra aj napriek svojej flexibilnosti a škálovateľnosti potrebuje menšie pamäťové zdroje ako ostatné prístupy (rozhodovací les, BV-TCAM). Využitie pamäte TCAM a analýza pravidiel dosahuje v niektorých prípadoch zvýšenie priepustnosti až o 76 % oproti základnej implementácii DCFL.

Bežným príkladom klasifikácie paketov je napríklad filtrovanie alebo monitorovanie IP tokov, ktorý vyžaduje iba vyhľadanie pravidiel na základe presnej zhody hodnôt. Filtrovanie paketov na základe IP tokov využíva takmer každé sieťové zariadenie a zároveň tento druh filtrovanie vyžaduje veľkú kapacitu množiny pravidiel. Vyhľadanie pravidiel na základe presnej zhody je menej komplexné ako ostatné typy vyhľadania, a preto je možné tento špecifický prípad vykonať efektívnejšie. Druhý navrhnutý prístup je špecializovaný práve na úplnú zhodu a optimalizuje klasifikáciu paketov založenú na úplnej zhode. Toto je dosiahnuté využitím distribuovaných pamätí na čipe FPGA na zrýchlenie algoritmu kukučieho hešovania. Vďaka distribuovaným pamätiam je možné vykonať v každom takte viac nezávislých vyhľadaní a tým pádom je možné zvýšiť rýchlosť vyhľadania aj minimálnou duplikáciou pamätí. Architektúra má teda nízke pamäťové nároky aj pri škálovaní na vysoké priepustnosti. Architektúra je schopná dosiahnuť priepusnosť viac ako 2 Tb/s

s efektívnou kapacitou viac ako 40 000 pravidiel reprezentujúcich IPv4 toky za cenu len niekoľko stoviek blokových pamätí (366 BRAM na čipoch od firmy Xilinx, 672 M20K na čipoch od firmy Intel).

Navrhnuté architektúry využívajú pamätové bloky dostupné priamo na čipe, čo limituje kapacitu pravidiel klasifikácie. Monitorovanie veľkých sietí vyžaduje rádovo milióny pravidiel. Na dosiahnutie potrebnej kapacity pravidiel je potrebné využiť externé pamäťové bloky. Externé pamäťové bloky prinášajú zvýšenú komplexitu prístupu do pamäte, najmä v prípadoch, keď dochádza k úpravám hodnôt v externej pamäti. Aby bola zabezpečená konzistencia a správnosť prístupov do externej pamäte, muselo by spracovanie paketov často čakať. Navrhnutá architektúra vyrovnávacej pamäte dovoľuje efektívne využiť externé pamäťové bloky. Architektúra dosahuje 3-krát vyššiu priepustnosť oproti iným mechanizmom a umožňuje spracovanie paketov na plnej rýchlosti 100 Gb/s. Zároveň architektúra znižuje počet potrebných skutočných prístupov do externej pamäte.

Využitím navrhnutých architektúr je možné dosiahnuť škálovateľnú a efektívnu klasifikáciu paketov na vysokorýchlostných sieťach. Architektúry sú zároveň mapovateľné z popisu v jazyku P4, čo prináša doposiaľ nevídanú flexibilitu.

Mapping of packet processing from P4 Language to FPGA Technology

Declaration

I hereby declare that this PHD thesis was prepared as an original work by the author under the supervision of Ing. Jan Kořenek, Ph.D. My colleagues from Netcope and Intel were helpful as well. I have listed all the literary sources, publications, and other sources, which were used during the preparation of this thesis.

.....
Michal Kekely
October 30, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Information technologies are constantly progressing, and the need for more and faster connections between computational devices and users is increasing. The Internet is getting bigger, and it supports a spectrum of different services. Computer networks and their infrastructure are required to be constantly faster and faster as users want to transfer more data. There are more network devices that communicate over the network links, and data-intensive communication is becoming more frequent, for example, video or audio streaming and sharing of various other media. This trend requires devices to have more computational power to be able to keep up with the amount of data transferred. The ever-increasing capacity of network links leads to a need for all network devices and systems to speed up their packet processing. Finally, the whole character of the networks is changing as well. The end devices are becoming more mobile, which leads to more dynamic networks.

Some applications require processing power that standard CPUs don't provide. While the software approach provides a lot of flexibility and ease-of-use the performance and efficiency are limited. An alternative is to use hardware acceleration, where the critical parts of the computation are offloaded into specialized hardware modules. Leveraging the hardware allows for more parallelization, pipelining, and the modules can be optimized with the desired functionality in mind. Hardware solutions can bring significant speed up and decrease energy consumption. A downside is that hardware modules are specialized and single-purpose. Therefore, flexibility suffers. Designing and implementing hardware modules is also way more complex and time-consuming.

In order to achieve wire-speed processing with a throughput of 100 Gbps and more, network systems have to utilize hardware-accelerated FPGA or ASIC technology. The FPGA acceleration provides high performance and is highly configurable (flexible compared to ASIC) as well. Flexibility is essential for any practical network system because traffic processing is changing with the introduction of every new protocol, application or service. Therefore, 40 Gbps and 100 Gbps network interface cards with FPGAs (also known as FPGA Smart NICs) started to be recently deployed to data centres as hardware platforms for acceleration [10] and will probably be more and more frequently used in the future.

The increasing complexity and computational requirements of computer networks led to the idea of separating the more complex control plane and the more performance-heavy data plane. The concept is also called Software Defined Networking (SDN) [47]. The idea can be pushed even further, where flexible network traffic processing can be easily described in the P4 high-level language [8]. The P4 language was originally designed at Stanford University in order to enable protocol, vendor and target independent definitions of packet processing. One of the integral parts of the P4 language specification [50, 51] is the

utilization of match/action tables as a basis to control the processing of each input packet. Furthermore, this description can be automatically mapped directly to a high-throughput packet processing architecture for an FPGA hardware accelerator [6, 7].

The core functionality performed by the match/action tables is packet classification in various forms. During the classification process, packets are matched against a rule set, which are usually defined by exact values, ranges or prefixes of a few selected packet header fields. Generally, the performed classification is a mathematical problem of a multi-dimensional range search. Packet classification is a core functionality for any network function, from switching and routing to network monitoring and security. It allows to change behaviour on a per-packet basis. Due to the large ruleset size and complexity of rules, it is rather difficult to perform matching at such rate that is sufficient for wire-speed processing of high-speed network data. Therefore, many different hardware architectures have been designed to accelerate packet classification [48, 17, 43, 34, 12, 41, 31].

In order to achieve wire-speed 100 Gbps throughput, it is necessary to process every incoming packet only in 6.7 ns because the shortest 64 B Ethernet packets can arrive within such time intervals. The time to process a packet corresponds to a 150 MHz clock. This time is even shorter for 400 Gbps and 1 Tbps, where we would need to scale the frequency to be over 500 MHz. FPGAs are not ready for such high frequency, especially if large data widths are used to transfer packet data. Therefore, the processing throughput is usually increased simply by the utilization of multiple processing pipelines in parallel [33, 42], which require multi-port memories or memory replication. Unfortunately, both approaches significantly reduce throughput scalability at 400 Gbps or 1 Tbps fast links.

1.1 Objectives

This thesis focuses on researching the area of packet classification in FPGA. As previously mentioned, packet classification needs to be efficient and scalable enough to be able to handle the increasing link speeds. On the other hand, the approaches also need to be flexible and general enough to handle different use cases with different amounts of available resources. Therefore, the research steps are taken with an emphasis on the generality, scalability and flexibility of designed algorithms. Since the FPGA resources are limited, and a lot of them might be used by other parts of packet processing, the algorithms also need to be as resource-efficient as possible. The research aims to satisfy at least 100 Gpbs links. The main goal is to leverage FPGA performance for accelerating packet classification flexibly and enabling it to be used via easy-to-use P4 language description.

Based on the mentioned focus in the area of hardware accelerated packet classification and P4 language, the following working hypothesis was formulated for this thesis:

Despite current approaches to packet classification not being sufficient in terms of required throughput and flexibility required to classify packets based on different dimensions, using the P4 language and new hardware architectures, it is possible to achieve high throughput (100 Gbps, 200 Gbps, 400 Gbps) while maintaining high flexibility.

The main objective of the thesis can be divided into multiple sub-tasks:

1. Analyze different approaches to packet classification. The main focus of the analysis should be on the feasibility of hardware implementation. Additionally, the different characteristics of each approach should be analyzed.

2. Identify the best approaches for packet classification in FPGA. Create a set of different approaches for different types of packet classification.

3. Design and implement selected approaches. The approaches should aim to satisfy a throughput of at least 100 Gbps while being efficient in terms of chip resources.

4. Evaluate the throughput, scalability, flexibility and efficiency of implemented approaches.

5. Analyze the rule sets used for different use cases. The goal is to optimize and increase the efficiency of selected approaches even further.

6. Design and implement an efficient way of mapping the P4 described packet classification (match/action tables) onto a specific configuration of the designed hardware approach.

P4 language can describe any match/action table. The tables can have different capacities (sizes of rule sets they can hold). The matching can be done based on different numbers of packet header fields (dimensions), where each field might be matched using different matching type (exact, ternary, longest prefix match). Different use cases might also require different throughput. Additionally, resource requirements should be considered. Real use cases require multiple match/action tables. For general packet processing, the packet also needs to be parsed, modified and then put together. Hardware modules for different functions and multiple match/action tables need to all fit into the limited FPGA resources. In order to properly analyze different approaches, there are multiple characteristics that should be considered. Based on the focus of the thesis, the main characteristics to consider are:

- **Speed of look-ups**. This characteristic is becoming more and more relevant. Even for 10 Gbps, the edge case of 64-byte long TCP packets equals to the processing of 20 million packets every second.

- **Low memory requirements**. Low memory requirements enable the usage of faster and smaller memories, which increases the throughput. Lower memory requirements also give more space for other functionality on the chip.

- **Support for different sizes of rule sets**. More complex and bigger networks require bigger rule sets. This characteristic usually goes hand in hand with memory requirements.

- **Supported types of matches**. Different use cases need different types of matching (exact, ternary, longest prefix match).

- **Support for different number of dimensions**. Dimensions are different packet header fields that are used for matching. Some approaches do not scale well when increasing the size or number of those fields.

- **General scalability of other characteristics**. Different use cases might need different numbers and sizes of dimensions, different sizes of rule sets, and different throughputs, and they might impose different area restrictions. The approaches should most efficiently fit the specific use case.

1.2 Outline

This thesis is written as a collection of papers. Research contributions are presented by a set of selected peer-reviewed papers and journal articles. The texts are provided at the end of the thesis in their original format in appendix A. The text of the thesis itself is organized into three chapters. Chapter 2 contains a summary of the relevant current state of the art. It describes basic network principles, Software-Defined Networking (SDN), P4 language and approaches used for packet classification. Chapter 3 describes the research and its main contributions. It also contains overview of selected included research papers. Chapter 4 summarizes the results and conclusion of the thesis. It also provides some discussion and possible future directions for the research.

Chapter 2

Current State of the Art

Computer networks are telecommunication networks that allow devices to exchange data. The two devices that communicate (sender and receiver or also end devices) are usually not connected directly but through a network of intermediary devices. Intermediary devices ensure that the message is routed and delivered to the end device.

For two devices to properly communicate, they both also have to understand each other. The data and metadata (data that are attached to the original data and carry some additional information about the data, for example, length of the message, address of the sender or time when the data were received) exchanged by the devices need to have a meaning attached. Network protocols define the structure and meaning of different parts of data. The protocols can also define specific sequences of messages or mechanisms for managing the data exchange itself (for example, the size of messages). The most commonly used network protocols are Ethernet, IP (versions 4 or 6) and TCP or UDP.

To transfer data between two devices, the network devices usually hold routing information which describe where different traffic should be sent. Routing information can be represented, for example, by a table of IP addresses and corresponding actions for traffic coming from or bound for those addresses. The most basic action is to forward the packet to a specific network interface (where another intermediary or end device is connected). Each intermediary device needs to be properly configured. The standard way of doing this was configuring each device independently. For more complex networks, this becomes difficult since there is a lot of overhead and redundancies for keeping policies consistent across the entire network. Another problem is that the networks are not dynamic this way and depend on the devices used and manufacturers of those devices.

Software-Define Networking (SDN) [47] is a networking architecture where the control plane is separated from the data plane. The intelligence and state of the network are centralized, and the actual infrastructure is abstracted for the applications using it. SDN allows the network to be more programmable, automatized and controlled. It also enables more scalable and flexible networks to be built.

The motivation behind the inception of SDN is the necessity to have the network architectures that are suitable for today's complex networks. Standard computer networks are hierarchical, built from layers of switches connected in a tree-like structure. This hierarchical structure is efficient for a client-server type of communication. However, current networks are more dynamic (data transfers within data centres, distributed computation). Trends leading up to SDN were mainly that the patterns of use for the networks are changing, there are more and more mobile devices, and the amount of data transferred is increasing. These trends exposed the limitations of standard networking technologies. Standard

technologies were not designed with these use cases in mind. The main problems are inconsistent policies, non-scalability and dependency on device manufacturers.

The main idea of SDN is to separate the control of network infrastructure from the functional elements that route the data. Additionally, the architecture enables direct programming of the network control. The separation of control from individual network devices introduces abstraction on top of the functional infrastructure. Network applications and services can view the entire network as a single logical unit interconnecting the end devices. The function of an abstracted network is also easier to automatize, scale and is overall flexible.

SDN is illustrated by figure 2.1. The architecture is logically separated into three layers. Physical devices create the infrastructure of the network, and on top of them is an abstract control layer. The control layer represents the network intelligence, centralized in software SDN controllers that enable a global view of the network. Applications in the application layer then treat the network as a single logical switch. The network can also be managed from a single point, independent of specific requirements and attributes of devices in the infrastructure. Design and management of such a network become simple. The infrastructure devices can also be simpler as they only need to understand the instructions from the SDN controller.

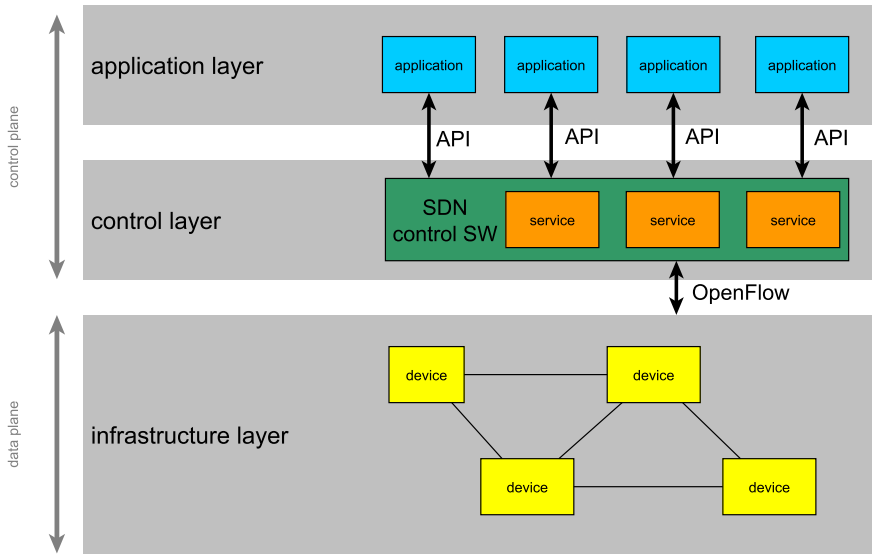

Figure 2.1: Architecture of SDN.

SDN also introduced the OpenFlow protocol. OpenFlow is a standardized communication interface between the control layer and the infrastructure layer. OpenFlow enables direct and uniform access to network devices (switches, routers). The protocol is necessary to separate network control of devices from centralized software. OpenFlow specifies basic primitives used by the external software applications for the configuration of routing of the network devices. The protocol is implemented and supported both in the devices and within the control software. A concept of network flows is used. A flow represents packets that belong to one logical network connection, so usually, they have the same 5-tuple of source IP address, destination IP address, source port, destination port and protocol. OpenFlow

uses a set of static and dynamic rules based on flows to define the behaviour of the network. The granularity of using flows enables fine control of the network.

SDN and OpenFlow are deployed on real networks, for example, by Google. The technology can be deployed in parallel with traditional routing, which makes it easier to switch to SDN. SDN can also manage only a part of the network. Therefore, not every device has to be SDN compatible.

There are downsides to SDN. While it is more flexible than standard networks, it is not flexible enough. Emerging new protocols (for example, different encapsulation protocols) introduce more and more header fields that can be used to classify packets. The issue is that OpenFlow only supports a set of header fields that can be used and need to be updated. OpenFlow needs to be constantly updated to support new fields. New versions of the protocol are incompatible with older ones and with older devices. Overall, the process of extending OpenFlow and updating all of the devices is slow and cumbersome.

2.1 P4 Language

The introduction of new network protocols requires the architectures and protocols like SDN and OpenFlow to be constantly updated and extended. Originally, OpenFlow was an abstraction of a single table of rules with 12 different header fields. The abstraction was extended over time. OpenFlow version 1.4.0 [4] supports 41 different header fields. This trend does not seem to stop anytime soon. Instead of extending OpenFlow, it is better to use more flexible mechanisms for packet analysis and header field matching.

P4 language [8, 50, 51] is meant to be used to configure network devices. The source code written in P4 language describes how the packets should be processed. This enables an even higher level of abstraction for programming computer networks. P4 language is highly domain-specific, and the expressive power is limited to the network functionality. Implementing support for different functions across different network devices would not be feasible.

P4 language has three main goals:

- **Reconfigurability**. The controller should be able to change the packet processing on the fly.
- **Protocol independence**. The device should not be dependent on any concrete packet format. The controller should be able to specify any protocol, any packet parser, and any set of lookup tables.
- **Target architecture independence**. The programmer of the controller needs no knowledge of the target architecture. The P4 compiler should be able to accept any target-independent description and translate it to a target-dependent configuration of the device.

2.1.1 Abstract Switch Forwarding Model

The P4 language works with an abstract model of a general switch device (figure 2.2). The switch starts by analyzing the packet headers and parsing them out. Based on the parsed values of header fields, packet processing is carried out as a sequence of Match/Action tables. The model contains a programmable parser, which gives it more flexibility than OpenFlow. Match/Action tables can be applied in parallel, and the actions that modify the packet

are constructed from primitives that are supported by each switch device. The model is a generalization of how packets are processed on different devices and technologies. Two types of operations are used for managing the model. Configuration operations programme the parser, sequence and control flow of applied Match/Action tables and the specific format of protocol headers. Runtime operations are used to fill the Match/Action tables with specific rules.

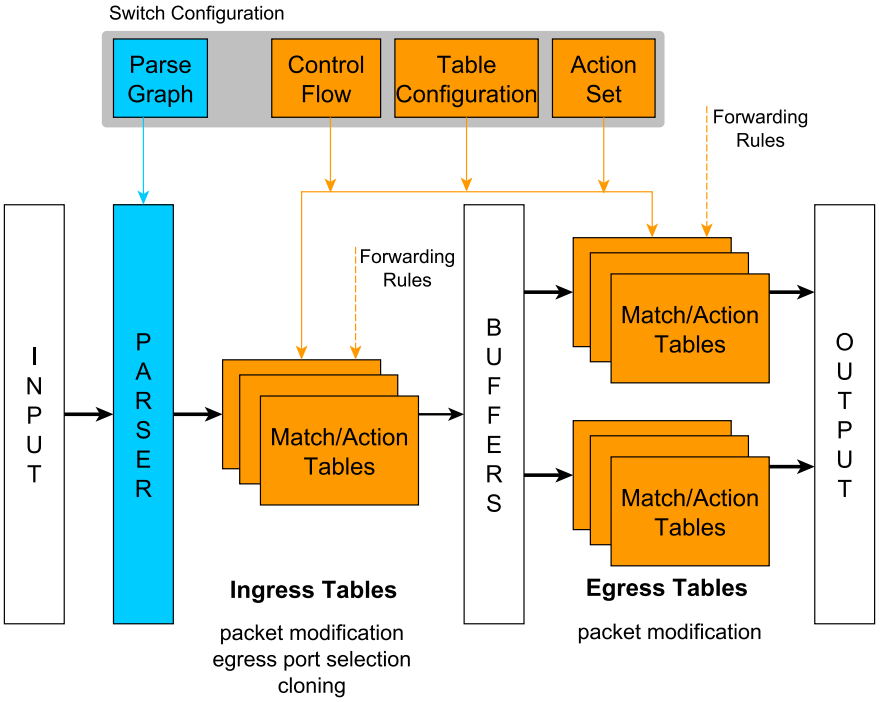

Figure 2.2: Abstract switch forwarding model of the P4 language.

Input packets are parsed by the parser, which extracts specific protocol headers and their fields. Parser ignores any meaning of the protocols or fields. Extracted header fields serve as input for the Match/Action tables. Each Match/Action table represents a rule set used to process the packets. The tables are separated into two groups, ingress and egress tables. Ingress tables process all of the incoming packets and can modify the packets and decide the output port of the packet or its priority. After ingress processing, the packets are placed into buffers that can implement queueing mechanisms. At this point, the output port for a packet is strictly set. Egress tables can further modify the packet for a given output port. Each rule in the Match/Action table specifies an action that will be applied. The action can modify header fields, modify packet metadata, duplicate the packet or drop it. Metadata are used to carry some of the additional packet information. The most commonly used metadata are the input port number of the incoming packet or the timestamp of arrival time.

2.1.2 Match/Action Tables

The P4 language is defined over the abstract model. The language contains basic concepts for defining the format of protocol headers and metadata, parse graphs, Match/Action

tables, actions based on primitives, and control flow. Control flow dictates the sequence of Match/Action table applications. Values from protocol headers and packet metadata are always tied to a single packet and do not persist between multiple packets. The P4 language supports definitions of stateful registers, counters and meters, which hold information that persists between packets. Accessing those stateful resources might be limited to specific tables.

Match/Action tables represent the classification rule sets. The Match/Action table description contains a set of header fields that are used for matching (classification dimensions). A type of match is also defined for each header field. The description specifies a set of actions that can be applied to modify the packet and maximal size of the table. An actual implementation of the table is never specified, and it is fully in control of the P4 compiler. There are four basic types of possible matches:

- **Exact match**. The compared value of the header field and reference value have to be the same.
- **Range match**. The rule defines a range of values. The header field is matched in terms of belonging to a given range.
- **Ternary match**. The rule defines a value and a mask. Bitwise AND operation is carried out on the header field before it is compared to the reference value. This effectively means that some bits of the value can be ignored for each comparison.
- **Longest prefix match (LPM)**. A rule with the longest matching prefix with the header field value is looked up.

The P4 language is becoming more and more popular all over the networking world. The language is currently being adopted by many companies, including Google and Intel.

2.2 Packet Classification

The P4 language tries to unify the programming of OpenFlow or other programmable switches. The core part of the language are Match/Action tables, which represent the standard packet classification problem [14, 54, 15].

The goal of packet classification is to assign packets to specified classes that are represented by a rule set (R, also called classifier). More specifically, a rule that corresponds to the packet is looked up in the rule set. To determine which rule corresponds to the packet, a value from packet header field is compared to values defined by the rules. Usually, the comparison is exact, ternary (using masks) or based on the longest matching prefixes. Based on the result of the classification, a specific action can then be carried out for the packet, for example, forwarding or dropping the packet.

The basic principle of packet classification can also be extended to matching based on multiple packet header fields. Each field represents one dimension of the classification, and it is also called multi-dimensional classification. The classification rule is represented as an n-tuple of values that are used for matching. Based on the number of classification dimensions, we can split the algorithms into single-dimensional and multi-dimensional.

2.3 Related Work

There is a lot of published research in the area of packet classification, with many completely different approaches described in individual papers. Some of them focus on being as general as possible, supporting packet classification in multiple different dimensions or supporting different types of match strength. However, the only way to scale most of the published approaches for higher throughputs is to utilize multiple copies of the same architecture operating in parallel. The problem of effective scaling is not properly addressed.

The most basic example of rule lookup is a linear algorithm. The algorithm goes through all the rules sequentially, comparing them to the packet. The algorithm either stops when it finds a matching rule (also generally called a table hit) or when all of the rules were compared and none of them matched the packet (also generally called a table miss). The linear algorithm, while being simple, is feasible only for small rule sets. Time complexity of the approach is *O(N*D)*, where N is the number of rules, and D is the number of dimensions.

In many cases, exact match packet classification is sufficient. This is prevalent mainly when IP flows are concerned. Effective approaches to exact match packet classification are usually based on hash tables. The hash table is a simple continuous table that is addressed by the result of a hash function computed on the looked-up value. A more sophisticated way of implementing hash tables is cuckoo hashing principle [39]. The main idea of cuckoo hashing, illustrated in figure 2.3, is to increase the efficiency of memory utilization in the hash table by multiple parallel hash functions/tables. Each table uses one of the different hash functions for indexing. This means that if a new element cannot be inserted into the first table because of a conflict with an already existing item, it can still be inserted into one of the other tables through a different hash function. Even when the element cannot be inserted into any of the tables, it can still be inserted by force, pushing out one of the previous occupants. The previous occupant can then be reinserted into the tables in the same manner. In the worst case, the insertion time for a new value is logarithmic. Using more tables and reinsertions allows the cuckoo hashing to keep the high lookup speed while decreasing the number of unresolvable conflicts and increasing the effective capacity.

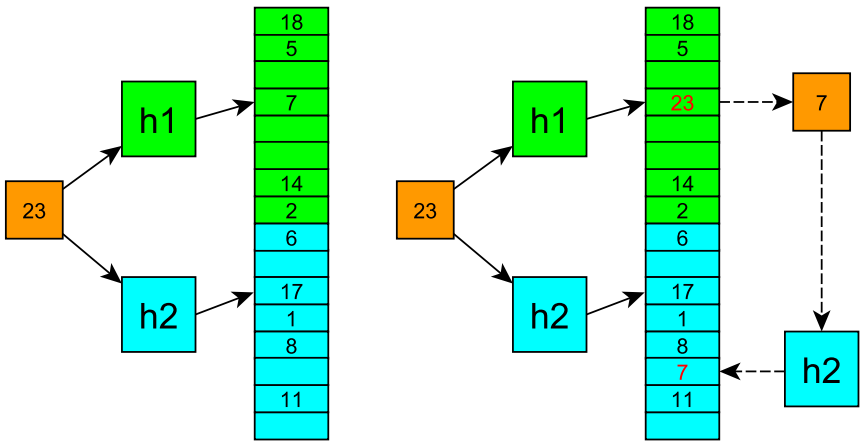

Figure 2.3: Basic idea of cuckoo hashing and adding a value into the table.

The cuckoo hashing approach is well-suited for hardware because each hash table can work in parallel [29, 22]. The hardware implementation can also leverage a small extra stash memory that keeps any values that were pushed out of the tables or values that could not be inserted due to a circular conflict. The stash memory is also matched and works in parallel with hash tables. Using the stash increases the effective capacity of the approach. These published implementations offer throughputs up to only 100 Gbps, with no proposed way to increase it. Cuckoo-hash-based packet classification is also effectively used to monitor or analyze network traffic with the idea of Software-Defined Monitoring (SDM) [21]. Here, external memory is utilized, and the achieved throughput is again shown to be sufficient only for up to 100 Gbps.

Packet classification based on bit-parallelism (or bit vectors, BV), proposed by Lakshman et al. [32], is a practical implementation that leverages the fact that rule updates are infrequent compared to search operations. The algorithm works in two stages. In the first stage, parallel searches are carried out within each of the dimensions, resulting in bit vectors. Each bit of these vectors corresponds to one record in the classification ruleset, therefore, their width is given by the number of rules used. A bit is set to a logical one if a corresponding rule is matched in a given dimension and is reset to logical zero otherwise. After the first stage, every bit vector represents the set of all the rules matched in one dimension. Then, the second stage has to find an intersection of the sets created within single dimensions. Since these sets are represented as bit vectors finding the intersection is reduced to a bitwise AND operation among the bit vectors. The main problem with the approach is scalability, as the width of bit vectors increases with the number of rules. The HW implementation would need to dynamically change when adding rules.

Ternary content-addressable memory (TCAM) is another hardware-suitable implementation. TCAM is capable of storing and comparing values along with the bit mask. The bit mask specifies which bits are used for the comparison. The hardware implementation of TCAM also allows for all of the stored values to be compared to the looked-up value in parallel. This means that the entire lookup can be done within a single clock cycle. The downside is that the parallel ternary lookup is costly and TCAMs tend to have limited capacity. Increasing the capacity of TCAM exponentially increases the logic resources required to implement the parallel lookup. Generally, only small TCAMs are used in combination with different algorithmic approaches.

Song et al. [44] presented architecture that combines bit vector approach with TCAMs. The architecture uses TCAMs for lookups within dimensions that require exact or prefix matches and tree-bitmap implementation of the BV algorithm for source and destination port lookups. This architecture is optimized for classification based on network flow 5-tuples, and therefore, it is not very flexible and was not shown to have the ability to scale to support different header fields.

Trie, shown in figure 2.4, is a binary tree data structure that is suitable for single-dimensional classification. Values in rules are converted into bit prefixes, and each prefix is represented by a path within a binary tree. Value from a packet is processed bit by bit. The value of each bit decides if the computation continues within the right or left subtree. The process is repeated until a leaf node is reached. Each node (or rather the entire path into the node) represents the longest-matched prefix. Adding and removing rules means adding or removing nodes representing the corresponding prefixes. The basic binary trie is not very efficient for wide dimensions. For example, for IPv6 addresses that are 128-bits wide, the matching may require up to 128 steps.

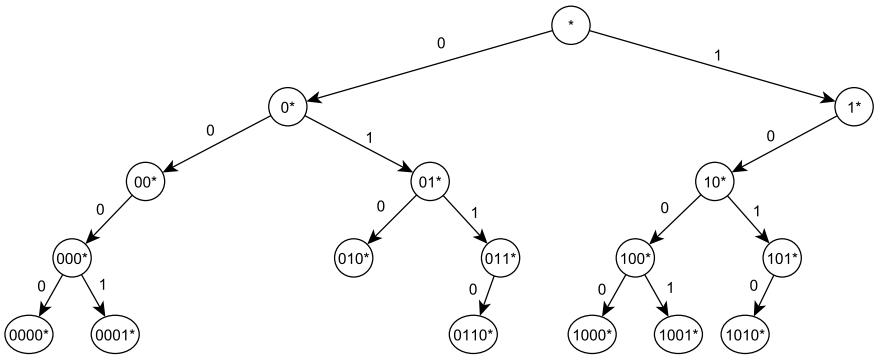

Figure 2.4: Example of a binary trie structure.

One of the possible solutions that make binary tries more efficient is to extend it to general n-ary (or multibit) trie. Each cycle looks at n bits at once. The number of bits used n is also called the stride. While this increases the lookup speed, it also increases memory requirements as each node of the tree is bigger, and the prefixes have to be aligned (or extended) to n bits. Controlled prefix expansion [45] uses multibit tries with additional memory optimizations. For example, moving the prefixes between leaf nodes (leaf pushing). Lulea algorithm [11] further improves the memory efficiency by compressing the data using bit maps. LC-Trie algorithm [38] selects different strides for different nodes, which ensures that no extended prefixes exist. Tree Bitmap [13] is comparable to Lulea. Additionally, it provides the ability to quickly insert and remove rules. The main idea of the algorithm is that each node has two functions. One is to guide the lookup. The other one is to transfer information about the rule that was found. Each node uses two bit maps, one for storing internal prefixes corresponding to the node and the other for storing pointers to its successors. The number of pointers can be further decreased by storing successors of a single node continuously. Therefore, each node only holds a pointer to the first successor and a pointer to an array of transitions to the successors. Figure 2.5 shows the structure of constructed multibit trie and figure 2.6 shows the structure of the stored bitmaps of the root node. The algorithm has an efficient hardware implementation as it is easy to parallelize. The memory requirements can be decreased even further by introducing hash functions (Hash-Tree Bitmap [52]).

All of the trie-based algorithms have poor scalability for multiple classification dimensions. Building hierarchical tries means that the size, lookup time and complexity increase exponentially [16]. The lookup time can be decreased, but this introduces redundancies, which further increase the size of the tries.

Several different approaches supporting multiple dimensions are described in [46]. A grid of Tries extends standard Trie to two dimensions. However, it is not easily extensible for more dimensions than two. The general solution using cross-products is more promising, but with no further optimization, it utilizes too much memory, and the resulting cross-products are large. Hierarchical tries are an extension of tries that support multiple dimensions. The tries are constructed recursively. While the number of dimensions D is higher than one, we build a trie for one of the dimensions. For each prefix in this dimension, we construct an entire hierarchical trie with *D-1* dimensions. The complexity and size of the constructed tries go up exponentially with the number of dimensions. Tries are usually efficient only for one or two dimensions.

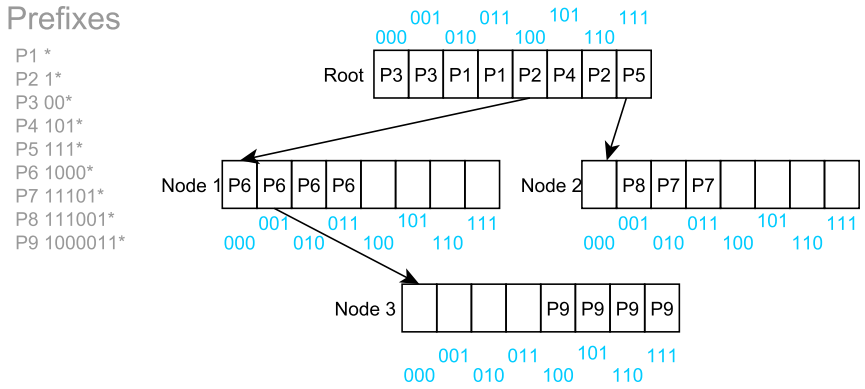

Figure 2.5: Example of constructed multibit trie.

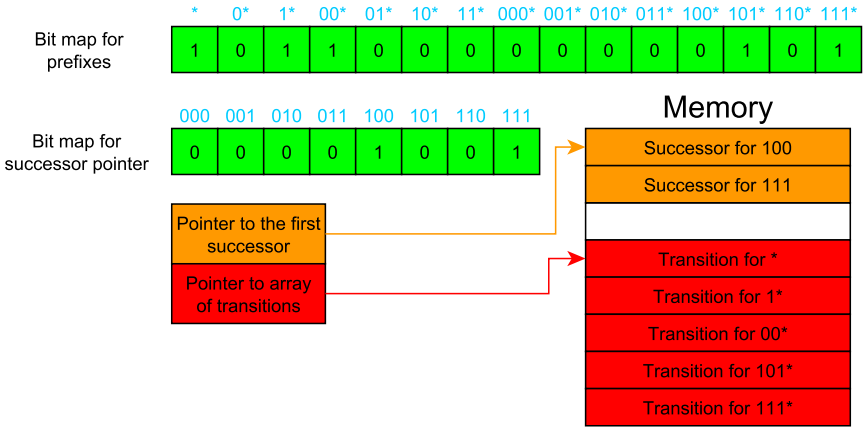

Figure 2.6: Structure of the stored bit maps of Tree Bitmap algorithm of root node from figure 2.5.

McKeown *et al.* [15] proposed using recursive flow classification (RFC). They suggest that packet classification can be viewed as mapping of N bits (given by the header fields) to M bits representing the rule or action matching given packet. Obviously M is expected to be way lower than N. Directly implementing such a mapping would require 2^N entries in memory. Therefore RFC algorithm splits this mapping into multiple stages that recursively map one set of values to a smaller set of values. The downfall of this approach is the memory requirements. Especially when the number of phases is low, the memory requirements are very high.

Another group of approaches utilizes architectures based on decision trees. The searched space can be represented geometrically, and the decision tree essentially recursively splits the space into smaller parts that are easier to search. The idea is illustrated by figure 2.7. Two-dimensional space is split into four parts by three cuts. Each rule can belong to multiple parts, for example, rule *R2* belongs to parts *2* and *4*.

HiCuts [17] and HyperCuts [43] are examples of hardware architectures based on these approaches. The main idea is to progressively cut the whole searched space represented by

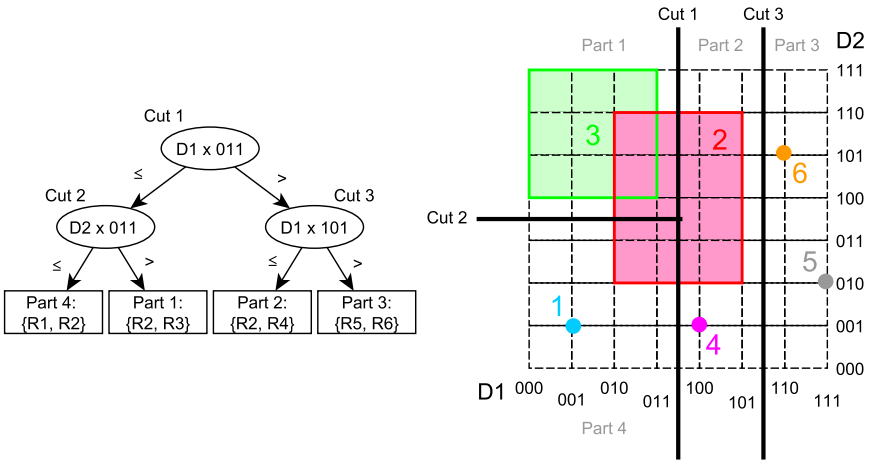

Figure 2.7: Example of search space cutting based on a binary decision tree.

the classification dimensions into small enough parts (usually representing one or only a few rules). Different heuristics can be used to decide how to cut the space efficiently, but resulting trees tend to have many nodes. Additionally, adding or removing rules leads to the need for rebuilding the whole tree. A way to increase throughput of HyperCuts was introduced by Luo *et al.* [35]. Their method, called explicit range search, uses new methods to cut ranges in dimensions and then search within the ranges. This leads to increased throughput for the price of needing to store explicit marks in memory. Kennedy *et al.* [28] implemented a simplified version of HyperCuts algorithm with the goal of reducing power consumption and increasing power efficiency. They were able to lower the frequency to only 32 MHz which however means a throughput of only 0.47 Gbps.

Prasanna et al. [20] pushed the idea of constructing decision trees even further. They have observed that HyperCuts and similar algorithms do not efficiently deal with rules that have too much overlap with each other. In such cases, many rules need to be duplicated, and the resulting tree (hence required memory) can explode exponentially with the number of dimensions. To mitigate this issue, a decision forest is introduced. A ruleset is split into subsets, and smaller decision trees are built for each of these subsets. Rules within each subset are chosen so that they have as little overlap as possible and that they specify nearly the same dimensions. Additionally, two other techniques are used to optimize HyperCuts algorithm. Rule overlap reduction stores rules that should be replicated in a list in each internal node instead of actually replicating it into all the child nodes. Precise range cutting is used to determine the cutting points which will result in the least number of rule duplications instead of deciding the number of cuts for a field.

Taylor et al. [48] introduced Distributed Crossproducting of Field Labels (DCFL). The algorithm is illustrated by figure 2.8. The basic idea is that the number of unique values that match the packet within a single dimension is low (usually up to five). The algorithm decomposes classification into single dimensions. In each dimension, a regular single-dimensional lookup is carried out, resulting in a set of unique values that match the packet. Each unique value is represented by a local label. The sets from each dimension are then aggregated together via a tree-like structure of aggregation nodes. In each node, a cross-product of two sets of partial results is created. The cross-product should be small

since the sizes of the sets are also small. Bloom filter arrays [12] are used to determine which elements of the cross-product are valid (part of some rule). Different aggregation nodes and single-dimensional lookups can run in parallel. Therefore, the algorithm is suitable for hardware implementation. Additionally, the approach is highly memory efficient and several parts of the architecture can be duplicated to increase throughput while still maintaining reasonable usage of on-chip memories and logic. The algorithm also scales well for any number of classification dimensions. The downside of the approach is that the throughput is dependent on the complexity of rule sets. Additionally, no efficient way of increasing the throughput for complex rule sets was shown.

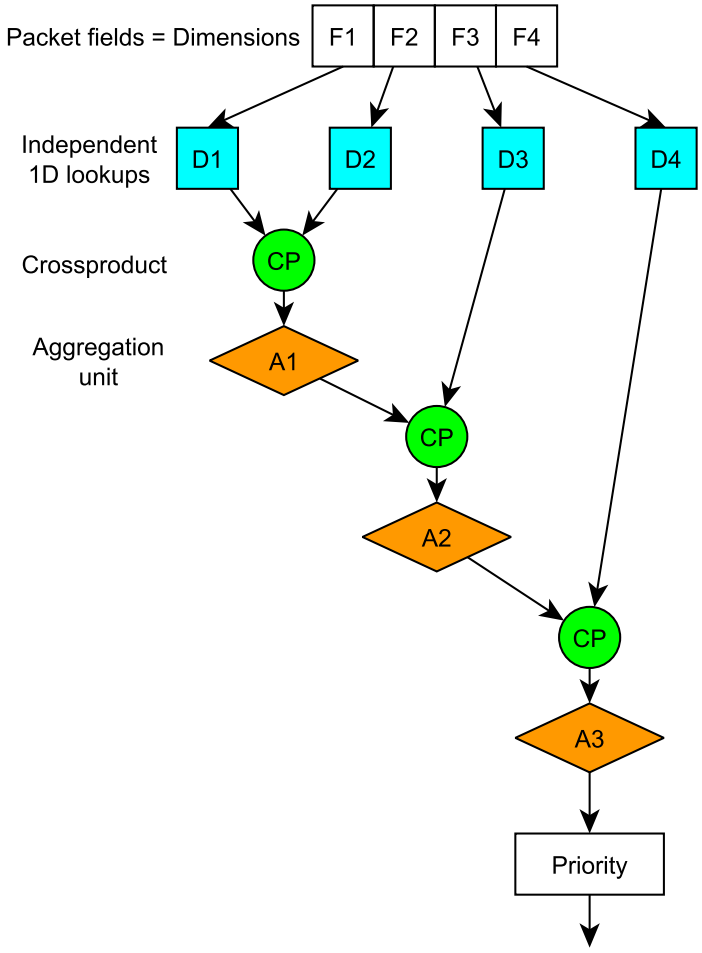

Figure 2.8: Architecture using DCFL algorithm.

The introduction of OpenFlow and later P4 language requires more flexible and scalable packet classification. The packet classification needs to be carried out for different use cases - multiple classification dimensions, different types of matches, different sizes of rule sets and different throughput requirements. The P4 language also allows a description of a series of multiple independent lookups, which need to be carried out for the same packet on the limited computational resources of the network device. Once again, each of those lookups might have different parameters. Therefore, all of the lookups need to be also

efficient. Most of the current approaches to packet classification are focused on solving only one particular use case. They are not flexible and scalable enough. The main issues are the bad scalability to multiple dimensions and unsupported types of matches, which makes using those approaches inefficient for more complex use cases. Another common issue is the inefficient use of memory, which leads to high memory requirements. This decreases the ability of those approaches to be used when a large number of independent lookups are needed. Finally, achieving high throughputs along with multiple lookups requires a level of hardware parallelization. Some of the approaches cannot be properly parallelized and have not been shown to have efficient hardware implementation.

Chapter 3

Research Summary

3.1 Research Process and Contributions

The current state of the art of packet classification, described in the previous chapter, is not sufficient for the flexibility and generality that the P4 language introduces and current high-speed networks. This leads to a need for more general, flexible and scalable approaches that enable efficient mapping from the P4 language onto the FPGA. The P4 language can describe any packet classification, and the target architecture (FPGA in our case) needs to be able to support it. P4 Match/Action tables have different sizes, different dimensions and different types of matches. Packet classification architecture has to address this flexibility but still maintain high performance.

Table 3.1: Worst case and mean case sizes of cross-products for different rulesets.

Ruleset	Cross-product 1		Cross-product 2		Cross-product 3		Cross-product 4	
	Worst	Mean	Worst	Mean	Worst	Mean	Worst	Mean
fw1_05_05	12	6	30	10	40	12	10	5
fw1_05_05	9	6	12	4	6	4	-	-
acl	54	46	148	8	148	12	-	-

Therefore, I started by designing general packet classification architecture. The best candidate for general packet classification is the DCFL algorithm. It splits the classification into single dimensions so it can support any combination of dimensions and match types. Thanks to the usage of probabilistic structures and a labelling technique, it is also highly efficient. Evaluating the results of the architecture shows that the bottleneck of the algorithm is the tree-like structure of aggregation nodes. Table 3.1 shows the sizes of cross-products for different rulesets. Checking a member of the cross-product requires one memory access, which means that big cross-products limit the achieved throughput, especially for complex rulesets with many overlapping rules.

The second step was to optimize the general packet classification architecture. The architecture needs to be fast and efficient. There are two techniques that can be used to increase the throughput of DCFL. Architecture using both of those techniques is further described in included papers [IP1] and [IP2] (appendixes A.1, A.2) and illustrated by figure 3.1. First, we can check the elements of cross-products faster. Duplicating memories of aggregation units increases the number of memory accesses that can be carried out each clock cycle. This means we can check multiple elements of the cross-product in parallel. We

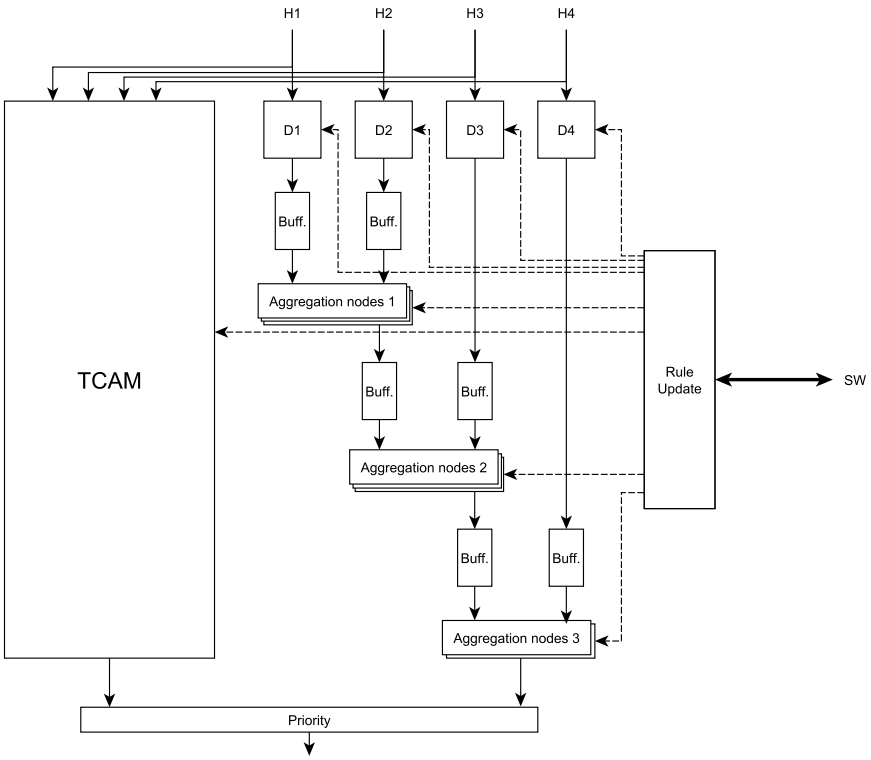

Figure 3.1: DCFL architecture using memory duplication and parallel TCAM.

can afford to duplicate memories because the memory requirements of aggregation units are already optimized by using Bloom filter arrays and the labelling of unique values. Memory duplication introduces the ability to find a balance between throughput and memory requirements. The second technique is to offload some of the rules into a small parallel TCAM. This technique aims to speed up the aggregation by decreasing the size of cross-products. The idea is that some rules (usually the most general ones) can match most packets. In other words, they contribute to the cross-product size the most. Removing those rules from consideration makes the algorithm work more efficiently. Additionally, this technique synergizes with memory duplication. Smaller cross-products require even less memory duplication.

The next step was to identify the most interfering rules correctly. A more detailed description of how rules can be identified and offloaded can be found in [IP6] (appendix A.6). The basic idea is to find unique values in classification dimensions that can be matched along the highest number of other unique values. These values are the ones that have the highest overlap with other values, and corresponding rules should be offloaded. The analysis of rules is done by building a trie structure from the unique values. Paths within the trie that go through the highest number of rules are then found.

The highly optimized and efficient DCFL architecture can be used for any general packet classification. If the use case requires only single-dimensional lookups or requires only exact type matching, it would not be efficient to instantiate the entire DCFL architecture when more efficient hardware architecture can be used for exact matching. For example, a common use case is filtering or monitoring of IP flows, which requires only exact matching.

This use case is used by most network devices and it requires a large rule capacity, and therefore, it is important to make it efficient. Each flow can be represented by one 5-tuple of exact values. Exact match is less complex than other types of matches, and therefore, the following step was designing an efficient architecture for this specific use case. The goal was to design an architecture that can be scaled to higher throughputs (in order of Tbps) and maintain low memory requirements. Such architecture is described in included papers [IP3] and [IP5] (appendixes A.3, A.5). The general Cuckoo hashing architecture can also be scaled up to higher throughputs by duplicating memories. However, due to the distributed nature of on-chip memories, we don't have to create as many duplicates. Tables with high rule capacity have their data spread across multiple distributed memory blocks. This means that even without any memory duplication, we can carry out multiple lookups as long as those lookups target entries in different memory blocks. By evaluating probabilities of collisions, we can determine how much memory duplication is required to achieve a given throughput. This technique allows us to scale throughput with minimal memory cost.

We cannot increase the capacity of tables enough by using only the on-chip memories, as those are limited. For example, mitigating DDoS attacks or monitoring big networks require millions of different flows to be matched (and therefore millions of exact match rules). External memories offer sufficient capacity. The issue is that accessing external memories may be costly. For example, updating stateful information (like a packet counter) in a Match/Action table requires the current value to be read out of the memory, updated and then written back (also called a read-modify-write operation). High and variable latency of external memories introduces various hazards in this case. These hazards can be read after write (RAW), write after read (WAR) and write after write (WAW). Multiple read-modify-write operations might be in progress at any time and keeping everything consistent becomes complicated. This may lead to packet processing pipeline stalls as it has to wait until some of those memory operations finish. Generally, caching mechanisms [53, 5] are used to address memory access issues. An alternative is to move the processing as close to the memory as possible [19, 18]. However, all of the current approaches either rely on the locality of the accessed data to be efficient or are not feasible for an FPGA implementation.

This led to another step in the research process. To increase the rule capacity of the packet classification, an architecture capable of efficient external memory access was designed. The architecture uses the caching mechanism described in the included paper [IP4] (appendix A.4). The architecture keeps track of running memory accesses and updates. Any new request from the packet processing pipeline is added to the cache or updates an existing entry. The cache carries out the actual communication with external memory and the processing pipeline is not blocked. The cache also aggregates any updates and accesses to the same memory element, which lowers the number of external memory accesses and ensures consistency. Using the caching architecture extends the packet classification and increases rule capacity. It also provides a convenient way for storing and updating stateful information.

Finally, different optimized architectures for different types of packet classification needed to be properly utilized. Basic implementation of a P4 compiler is available within the p4c project [2]. The project provides an implementation of the front-end, some mid-end optimizations and example back-ends of the P4 compiler. Since all of the above-mentioned architectures are configurable and scalable, the main goal of the P4 FPGA back-end is to select which implementation is the most efficient for each table. The general rules for selecting tables are:

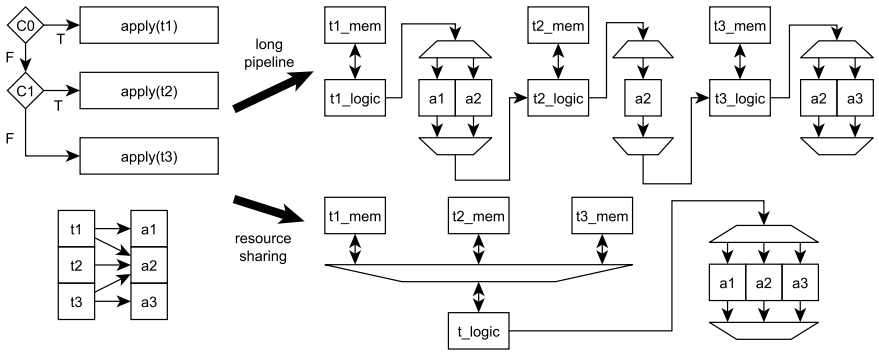

Figure 3.2: Resource sharing between tables.

- Very small tables are implemented using simple parallel comparisons via on-chip logic and registers or TCAM.
- Exact match tables are implemented using optimized Cuckoo hashing architecture.
- Single-dimensional tables are implemented using the appropriate single-dimensional approach.
- More complex tables are implemented using optimized DCFL architecture.
- Very large tables additionally leverage the caching and external memories.

One last thing that can be considered in terms of optimizations is that in most cases the classification algorithm is typically a part of larger network application. This means that it is instantiated along with other instances of packet classification. Figure 3.2 illustrates an optimization when multiple P4 tables can share the same resources. There are three tables *t1*, *t2* and *t3*. These tables are applied in different branches of *if-else* statement (meaning they are applied exclusively). Table *t1* has actions *a1*, *a2*, table *t2* has *a2* and table *t3* has *a2* and *a3*. If we generated every table independently, we would end up with 3 tables and 5 actions. If we were to use resource sharing, we would only need to generate 1 table with 3 times as much memory and 3 actions. This can be even further optimized if the tables are rather small and do not take up an entire BlockRAM. In this case, the tables can even share the BlockRAM (as at any given time only one of them will try to access the memory). Assuming we have 32b wide BlockRAMs with 1024 entries and tables *t1* and *t2* require 512 entries of 16b wide data and table *t3* requires 512 entries of 30b wide data. We can easily fit memories for all of the 3 tables into a single BlockRAM - first 512 entries will contain 16b of *t1* data and 16b of *t2* data, while the other 512 entries will contain 30b of *t3* data. Some tables can even be merged together fully. For example, there are two P4 tables that filter IP flows, but one does so using 32b IPv4 addresses, while the other uses 128b IPv6 addresses. Since a packet cannot simultaneously have both versions of the IP header, we can merge these tables. The resulting merged table filters flows using 128 wide metadata fields. Values of the metadata fields are either set to the IPv6 addresses (for IPv6 packets) or IPv4 addresses are mapped onto the lower 32b of the metadata fields (for IPv4 packets).

To summarize, the research presented in this thesis is focused on the design, implementation and evaluation of packet classification on FPGA. The focus is on the classification

architectures being general, configurable and scalable. This allows them to be utilized by a P4 language compiler to generate a packet processing architecture.

3.2 Papers

This section goes over brief descriptions of the included papers. For each paper, a brief motivation is presented, followed by the key contributions to the research presented in the thesis. Full texts of included papers can be found in appendix A. Afterwards, a brief description of other papers is also mentioned.

3.2.1 Included Papers

Included Paper 1

Mapping of P4 Match Action Tables to FPGA
The full text of the paper [IP1] can be found in appendix A.1. Current approaches to packet classification for high-speed networks are not flexible and scalable enough for an efficient general mapping from the P4 language description. Network architects can describe any type and number of tables in the P4 language. This means that there has to be a general architecture that is capable of efficient packet classification for different numbers of dimensions and types of matches in those dimensions. Additionally, the resources on an FPGA are limited and for any reasonable packet processing the classification itself is not sufficient. This means that those already limited resources need to be used as efficiently as possible to accommodate all of the required functions and modules. Finally, there also has to be an architecture that can be scaled to high throughputs without sacrificing the above-mentioned qualities.

Key Contribution
The main contribution of this paper is an introduction of efficient hardware architecture for packet classification. The architecture can be scaled to 100 Gbps while maintaining feasible memory requirements. The architecture is general and supports any number of dimensions and all of the standard types of matches by leveraging optimal single-dimensional approaches for each. The proposed techniques and optimizations also give us the ability to balance between resource requirements and throughput. We can increase throughput by using more resources. The paper introduces the architecture, optimizations and the results that we were able to achieve. The paper also outlines how the intelligent mapping from P4 might look like.

Abstract
Current networks are changing very fast. Network administrators need more flexible and powerful tools to be able to support new protocols or services very fast. The P4 language provides new level of abstraction for flexible packet processing. Therefore, we have designed new architecture for memory efficient mapping of P4 match/action tables to FPGA. The architecture is based on DCFL algorithm and is able to balance the processing speed and available memory resources.

3.2.2 Included Papers

Included Paper 2

Packet Classification with Limited Memory Resources
The full text of the paper [IP2] can be found in appendix A.2. The goal of this thesis is to have packet classification that is flexible and efficient. Paper [IP1] only introduced the DCFL architecture and presented some basic results. However, the architecture needs to be further optimized and efficient for different use cases. Therefore, the main goal of this paper is to show more detailed optimizations and evaluate the architecture for more use cases.

Key Contribution
This paper extends and optimizes the hardware architecture from [IP1]. The architecture and optimizations are described in more detail, which allows the architecture to be used better. The paper also further evaluates results achieved by the architecture for more use cases. The main contribution of this paper is an efficient hardware architecture for packet classification that is shown to be efficient and scalable for different use cases of general packet classification.

Abstract
Network security and monitoring devices use packet classification to match packet header fields in a set of rules. Many hardware architectures have been designed to accelerate packet classification and achieve wire-speed throughput for 100 Gbps networks. The architectures are designed for high throughput even for the shortest packets. However, FPGA SoC and Intel Xeon with FPGA have limited resources for multiple accelerators. Usually, it is necessary to balance between available resources and the level of acceleration. Therefore, we have designed new hardware architecture for packet classification, which can balance between the processing speed and hardware resources. To achieve 10 Gbps average throughput the architecture needs only 20 BlockRAMs for 5500 rules. Moreover, the architecture can scale the processing speed to wire-speed throughput on 100 Gbps line at the cost of additional memory resources.

Included Paper 3

Memory Aware Packet Matching Architecture for High-Speed Networks
The full text of the paper [IP3] can be found in appendix A.3. There are use cases where exact match lookups are sufficient for a Match/Action table. For example, filtering based on the Software-Defined Monitoring (SDM). The Match/Action table is used to filter specific flows. This generally also means that the table requires higher capacity. Using DCFL architecture from [IP1] and [IP2] is not the most efficient approach. Existing approaches to exact match packet classification were already optimized but have not been shown to efficiently scale up to higher throughputs. This paper addresses the efficient scalability. The paper also won the **Best Student Paper Award** at the conference, it was presented on (DSD 2018).

Key Contribution
The paper introduces a novel way of scaling up the throughput of the Cuckoo hashing algorithm. The memory-efficient parallel hardware architecture leverages the distributed

FPGA memories to increase throughput without requiring as much extra memory. In terms of the research presented in the thesis, it enables even more efficient implementation and mapping of large exact match Match/Action tables. The concept of efficient and scalable packet classification, implemented in FPGA alongside other functions and modules, is strengthened even further.

Abstract
Packet classification is a crucial operation for many different networking tasks ranging from switching or routing to monitoring and security devices like firewall or IDS. Generally, accelerated architectures implementing packet classification must be used to satisfy the ever-growing demands of current high-speed networks. Furthermore, to keep up with the rising network throughputs, the accelerated architectures for FPGAs must be able to classify more than one packet in each clock cycle. This can be mainly achieved by utilization of multiple processing pipelines in parallel, what brings replication of FPGA logic and more importantly, scarce on-chip memory resources.

Therefore, in this paper, we propose a novel parallel hardware architecture for hash-based exact match classification of multiple packets per clock cycle with reduced memory replication requirements. The basic idea is to leverage the fact that modern FPGAs offer hundreds of BlockRAM tiles that can be accessed (addressed) independently to maintain high throughput of matching even without fully replicated memory architecture. Our results show that the proposed approach can use memory very efficiently and scales exceptionally well with increased record capacities. For example, the designed architecture is able to achieve throughput of more than 2 Tbps (over 3 000 Mpps) with an effective capacity of more than 40 000 IPv4 flow records for the cost of only 366 BlockRAM tiles and around 57 000 LUTs.

Included Paper 4

Pipelined ALU for Effective External Memory Access in FPGA
The full text of the paper [IP4] can be found in appendix A.4. One of the important parts of any packet classification approach is the capacity of rules that it can support. Architectures presented in papers [IP1], [IP2] and [IP3] use only on-chip memories. This limits the rule capacity that can be achieved. External memories need to be used to increase the capacity further. Accessing external memories on FPGA is generally not an easy task. The high and variable latency of external memories might lead to stalls in the entire processing pipeline. Additionally, nearly all network use cases require stateful information to be stored. For example, the number of packets and bytes is counted for specific flows. P4 language uses registers and counters for this. Storing the rules and stateful information in on-chip memories or logic is simple. The access to those memories is instant, and any stateful information updates can be done instantly. Variable and high latency of external memory create issues where the computation might stall even more to ensure consistency of updates of data in the external memory. Finally, access to external memories is a potential bottleneck because they are centralized and have a limited number of access ports. It is important to address all of those issues.

Key Contribution
The paper presents a design and evaluation of hardware architecture that optimizes external memory accesses. The architecture acts as a cache for accessing the external memory.

There are multiple benefits that this introduces. First, the number of stalls for the processing pipeline is greatly reduced. Second, the number of actual external memory accesses is reduced as well. Third, the architecture can aggregate and ensures the consistency of updates in external memory. The architecture enables efficient usage of external memories for storing packet classification rules and stateful information associated with them. Therefore, packet classification can support higher rule capacities (millions of entries), while maintaining full throughput of at least 100 Gbps.

Abstract

The external memories in digital design are closely related to high response time. The most common approach to mitigate latency is adding a caching mechanism into the memory subsystem. This solution might be sufficient in CPU architecture, where we can reschedule operations when a cache miss occurs. However, the FPGA architectures are usually accelerators with simple functionality, where it is not possible to postpone work. The cache miss often leads to a pipeline stall or even to data loss. The architecture we present in this paper reduces this problem by aggregating arithmetic operations into the memory subsystem itself. Fast data processing is achieved because arithmetic operations working with external data are offloaded. Our architecture reaches a speed of 200 Mp/s (operations carried out). It is designed to be used in systems with link speeds of 100 Gb/s. It outperforms other implementations by a factor of at least 3. The additional benefit of our architecture is reducing the number of memory transactions by a factor of two on real-world datasets.

Included Paper 5

General Memory Efficient Packet Matching FPGA Architecture for Future High-Speed Networks

The full text of the paper [IP5] can be found in appendix A.5. One of the main goals of the P4 language is the target architecture independence. Additionally, the goal of this thesis is to research, design and evaluate packet classification that is efficient for a broad spectrum of use cases. However, paper [IP3] was evaluated on Xilinx hardware only and for a limited scope of use cases. The exact match packet classification needs to be efficient for many different use cases. Also, possible further optimizations are needed. The goal of this paper is to provide a more detailed description and better evaluation of the results achieved.

Key Contribution

The key contribution of the paper is a better evaluation of optimized Cuckoo hashing architecture. The paper presents a broader and more in-depth analysis and evaluation of the proposed architecture. Results for Intel FPGA chips are also presented. This shows that the approach and optimizations are efficient on a wide array of FPGA chips and supports the target-independency notion of P4 language.

Abstract

Packet classification (matching) is one of the critical operations in networking widely used in many different devices and tasks, ranging from switching or routing to a variety of monitoring and security applications like firewall or IDS. To satisfy the ever-growing performance demands of current and future high-speed networks, specially designed hardware accelerated architectures implementing packet classification are necessary. These demands are now growing to such an extent that in order to keep up with the rising throughputs of

network links, the FPGA accelerated architectures are required to perform the matching of multiple packets in every single clock cycle. To meet this requirement, a simple replication approach can be utilized – instantiate multiple copies of a processing pipeline matching incoming packets in parallel. However, the simple replication of pipelines inseparably brings a significant increase in the utilization of FPGA resources of all types, which is especially costly for rather scarce on-chip memories used in matching tables.

We propose and examine a unique parallel hardware architecture for hash-based exact match classification of multiple packets in each clock cycle that offers a reduction of memory replication requirements. The core idea of the proposed architecture is to exploit the basic memory organization structure present in all modern FPGAs, where hundreds of individual block memory tiles are available and can be accessed (addressed) independently. This way, we are able to maintain a rather high throughput of matching multiple packets per clock cycle even without fully replicated memory resources in matching tables. Our results show that the designed approach can use on-chip block memory resources very efficiently and even scales exceptionally well with increased capacities of match tables. For example, the proposed architecture is able to achieve a throughput of more than 2 Tbps (over 3 000 Mpps) with an effective capacity of more than 40 000 IPv4 flow records at the cost of only a few hundred block memory tiles (366 BlockRAM for Xilinx or 672 M20K for Intel FPGAs) utilizing only a small fraction of available logic resources (around 68 000 LUTs for Xilinx or 95 000 ALMs for Intel).

Included Paper 6

Optimizing Packet Classification on FPGA

The full text of the paper [IP6] can be found in appendix A.6. As previously mentioned, the packet classification needs to be efficient. The chip area is shared by other packet processing (parsing, deparsing, complex actions, queuing, checksum computation) and possibly even other functions (for example, cryptography [P2]). Higher efficiency can be leveraged to scale the throughput or decrease memory requirements. The main goal of this paper is to make the DCFL approach even more efficient by optimizing the throughput bottleneck of the approach for complex rule sets.

Key Contribution

The paper further optimizes and evaluates the efficiency of DCFL-based architecture from [IP1] and [IP2]. Instead of changing the architecture, the optimization analyses the rule sets. This lowers the negative impact of some of the bottlenecks of DCFL and makes packet classification more efficient. Once again, increased efficiency can be translated into better scalability, better throughput or smaller resource strain. Overall flexibility increases even further.

Abstract

Packet classification is a crucial time-critical operation for many different networking tasks ranging from switching or routing to monitoring and security devices like firewalls or IDS. Accelerated architectures implementing packet classification must satisfy the ever-growing demand for current high-speed networks. However, packet classification is generally used together with other packet processing algorithms, which decreases the available hardware resources on the FPGA chip. The introduction of the P4 language requires the packet classification to be even more flexible while maintaining a high throughput with limited resources. Thus, we need flexible and high-performance architectures to balance processing

speed and hardware resources for specific types of rules. DCFL algorithm provides high performance and flexibility. Therefore, we propose optimizations to the DCFL algorithm and overall packet processing hardware architecture. The goal is to maximize the throughput and minimize the resource strain. The main idea of the approach is to analyze the ruleset, identify some conflicting rules and offload these rules to other hardware modules. This approach allows us to process packets faster, even in the worst-case scenarios. Moreover, we can fit more packet processing into the FPGA and fine-tune the packet processing architecture to meet a specific network application's throughput and resource demands. With the proposed optimizations we can achieve up to a 76 % increase in the throughput of the packet classification. Alternatively, we can achieve up to a 37 % decrease in resources needed.

3.2.3 Other Relevant Papers

Relevant Paper 1

Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput

Paper [P1] describes a hardware implementation of a general configurable packet parser. Packet parsing is a crucial operation that packet classification directly depends on. Efficient and scalable packet parser and packet classification represent a minimal implementation that is capable of processing packets (for example, filtering can be done using only those two functions). The results of this paper show that it is possible to have high-speed packet parsing on FPGA. In combination with the included papers, it is possible to have an efficient and scalable packet processing pipeline on FPGA. This all builds on top of the original papers that showed this for 100 Gbps [6, 7].

Abstract

As throughput of computer networks is on a constant rise, there is a need for ever-faster packet parsing modules at all points of the networking infrastructure. Parsing is a crucial operation which has an influence on the final throughput of a network device. Moreover, this operation must precede any kind of further traffic processing like filtering/classification, deep packet inspection, and so on. This paper presents a parser architecture which is capable to currently scale up to a terabit throughput in a single FPGA, while the overall processing speed is sustained even on the shortest frame lengths and for an arbitrary number of supported protocols. The architecture of our parser can be also automatically generated from a high-level description of a protocol stack in the P4 language which makes the rapid deployment of new protocols considerably easier. The results presented in the paper confirm that our automatically generated parsers are capable of reaching an effective throughput of over 1 Tbps (or more than 2000 Mpps) on the Xilinx UltraScale+ FPGAs and around 800 Gbps (or more than 1200 Mpps) on their previous generation Virtex-7 FPGAs.

Relevant Paper 2

200 Gbps Hardware Accelerated Encryption System for FPGA Network Cards

Paper [P2] shows a specific use case for FPGA packet processing. It describes an encryption system that is capable of high-speed (200 Gbps) network data encryption utilizing the IPsec protocol. This demonstrates the need for efficient packet processing since it can be used

as a basis for other functions that are outside of the scope of P4 language (in this case, cryptography).

Abstract
We present the architecture and implementation of our encryption system designed for 200 Gbps FPGA (Field Programmable Gate Array) network cards utilizing the IPsec (IP security) protocol. To our knowledge, our hardware encryption system is the first, that is able to encrypt network traffic at the full link speed of 200 Gbps using a proven algorithm in a secure mode of operation on a network device that is already available on the market. Our implementation is based on the AES (Advanced Encryption Standard) encryption algorithm and the GCM (Galois Counter Mode) mode of operation, therefore it provides both encryption and authentication of transferred data. The design is modular and the AES can be easily substituted or extended by other ciphers. We present the full description of the architecture of our scheme, the VHDL (VHSIC Hardware Description Language) simulation results and the results of the practical implementation on the NFB-200G2QL network cards based on the Xilinx Virtex UltraScale+ chip. We also present the integration of the encryption core with the IPsec subsystem so that the resulting implementation is interoperable with other systems.

3.3 List of Publications

Included Papers

[IP1] Kořenek, J. and Kekely, M. Mapping of P4 Match Action Tables to FPGA. In: *Preceedings of 27TH INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS*. Institute of Electrical and Electronics Engineers, 2017. DOI: 10.23919/FPL.2017.8056768. ISBN 978-90-90-30428-1. Available at: https://www.fit.vut.cz/research/publication/11551

[IP2] Kekely, M. and Kořenek, J. Packet Classification with Limited Memory Resources. In: *In proceedings 2017 Euromicro Conference on Digital System Design*. Institute of Electrical and Electronics Engineers, 2017, p. 179–183. DOI: 10.1109/DSD.2017.61. ISBN 978-1-5386-2145-5. Available at: https://www.fit.vut.cz/research/publication/11550

[IP3] Kekely, M., Kekely, L. and Kořenek, J. Memory Aware Packet Matching Architecture for High-Speed Networks. In: *Proceedings of the 21st Euromicro Conference on Digital Systems Design*. IEEE Computer Society, 2018. DOI: 10.1109/DSD.2018.00017. ISBN 978-1-5386-7376-8. Available at: https://www.fit.vut.cz/research/publication/11819

[IP4] Kekely, M., Hynek, K. and Čejka, T. Pipelined ALU for effective external memory access in FPGA. In: *2020 23RD EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN (DSD 2020)*. Institute of Electrical and Electronics Engineers, 2020, p. 97–100. DOI: 10.1109/DSD51259.2020.00026. ISBN 978-1-7281-9535-3. Available at: https://www.fit.vut.cz/research/publication/12450

[IP5] Kekely, M., Kekely, L. and Kořenek, J. General memory efficient packet matching FPGA architecture for future high-speed networks. *Microprocessors and Microsystems*. Elsevier Science. 2020, vol. 73, no. 3. DOI: 10.1016/j.micpro.2019.102950. ISSN 0141-9331. Available at: https://www.fit.vut.cz/research/publication/12138

[IP6] Kekely, M. and Kořenek, J. Optimizing Packet Classification on FPGA. In: *PROCEEDINGS 2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. Institute of Electrical and Electronics Engineers, 2023, p. 7–12. ISBN 978-83-503-3276-7. Available at: https://www.fit.vut.cz/research/publication/12805

Other Relevant Papers

[P1] Cabal, J., Benáček, P., Kekely, L., Kekely, M., Puš, V. et al. Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, 2018, p. 249–258. DOI: 10.1145/3174243.3174250. ISBN 978-1-4503-5614-5. Available at: https://www.fit.vut.cz/research/publication/11674

[P2] Martinásek, Z., Hajný, J., Smékal, D., Malina, L., Kekely, M. et al. 200 Gbps Hardware Accelerated Encryption System for FPGA Network Cards. In: *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 2018, p. 11–17. DOI: 10.1145/3266444.3266446. ISBN 978-1-4503-5996-2. Available at: https://www.fit.vut.cz/research/publication/12244

Chapter 4

Disscussion and Conclusions

This chapter discusses the research presented in this thesis and summarizes the achieved results. Based on this, conclusions are drawn, and possible direction for future work is outlined.

The thesis researches packet classification on FPGA with a focus on high performance and flexibility. First, an analysis of the current state of packet processing and different packet classification approaches was conducted. Obtained information is summarized in chapter 2. The analysis was used as a base for further research and design of new hardware architectures. These architectures were then further optimized. The goal was to make them scalable and memory efficient. A valuable piece of information for the optimizations was provided by an analysis of the rulesets and network data and how they overlap and interact with the proposed architectures. Afterwards, the approaches were extended to higher rule capacity by designing an architecture for efficient usage of external memories. All of the architectures were tested and evaluated. Real network traces used for this evaluation were obtained from the high-speed backbone network managed by Cesnet. Cesnet is the Czech National Research and Educational Network operator with an infrastructure consisting of multiple optical links with bandwidth up to 100 Gb/s. The research conducted to design, optimize and implement the architectures is presented in chapter 3 and in included published papers.

4.1 Results

In order to evaluate the proposed concepts, the hardware architectures were realized on FPGA chips. Multiple classification rule sets were used. Two network data traces captured from CESNET network were used: *meter1* and *meter4*. Both traces contain 1,000,000 packets (approximately 1 second of network traffic) captured during different periods of day. Their basic characteristics are shown in table 4.1. Throughput was also evaluated for the worst-case scenario of shortest possible packets (packets from real traces truncated to 64B). This section summarizes the results achieved.

Table 4.1: Basic characteristics of captured traffic traces from CESNET network.

Trace name	Packets	Bytes	Time period	Capture time
meter1	1 000 000	1 081 259 293	1.033s	11:00
meter4	1 000 000	791 590 133	1.489s	15:00

General Packet Classification

General packet classification is carried out by the optimized DCFL architecture [IP1][IP2]. In order to analyze how effective the approach is, the architecture was implemented using high-level synthesis. For the evaluation, we used a chip from Kintex-7 family of FPGAs. To demonstrate a variety of rulesets, three classification rule sets are used. These sets were generated by ClassBench [49] tool and are part of NetBench [40] framework. Table 4.2 shows the characteristics of the sets, main one being level of overlap between ranges or prefixes within rules. *acl1* is an example of ruleset with many overlapping ranges of port values, which leads to DCFL not being as effective. *fw2_05_05*, on the other hand, has little overlap between rules and prefixes, thus DCFL shows much better results. Finally, *fw1_05_05* represents a middle ground. The architecture runs at 200 MHz.

Table 4.2: Charecteristics of rulesets used.

Ruleset	Dimensions	Number of rules	Overlap
acl1	4	2406	high
fw1_05_05	5	733	medium
fw2_05_05	4	941	low

Figure 4.1 show scaling of block RAMs and logic needed for increasing throughput of the optimized architecture for rulesets *fw1_05_05* and *acl* compared to results for comparable configurations of other general approaches. There are two graphs - one shows the comparison and scaling of block RAMs, and the other the comparison and scaling of LUTs. The logic requirements of BV-TCAM and HyperCuts could not be reliably determined, therefore, they are omitted. Note that the architecture has capacity of around 5500 rules, whereas other architectures may have different rule capacities (mainly decision forest has capacity of 10 000 rules). Additionally, the architecture used quite small TCAMs (since number of unique values in single dimensions is a lot lower than number of rules) as engines for searching in some dimensions. We can see that the resource requirements stay manageable even for high throughputs. The optimized approach is comparable to or even better than other approaches in terms of block RAMs needed, while being also comparable in logic required. Since other apporaches have not been shown to be scalable and flexible, the other main benefit of the optimized approach is the scalability and flexibility that was already mentioned.

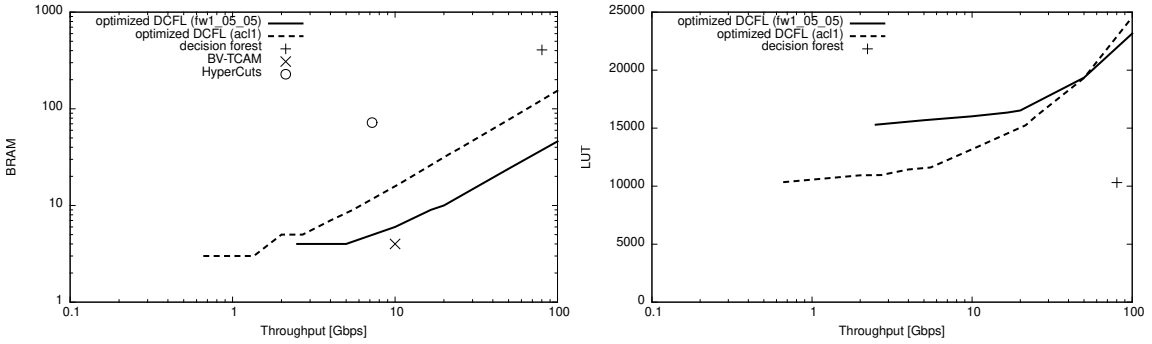

Figure 4.1: Block RAMs and LUTs used by optimized DCFL compared to other approaches.

Part of the scalability and efficiency is provided by memory duplication and Bloom filter arrays. However, offloading the most conflicting rules into a small TCAM with 32 entries also brings substantial results. Figure 4.2 shows the percentage increase in the throughput brought by offloading of the rules. Different heuristics to determine which rules should be offloaded are evaluated (as the full analysis might be complex and not always feasible). There are two different types of heuristics - on-the-fly analysis when the rules are analyzed one by one as they are being added to the packet classification and offline analysis, when the entire rule set is available prior to the packet classification. Each case also evaluates interference between rules (based on trie-based analysis described in [IP6]) or uses the generality of a rule to determine if the rule should be offloaded. Additionally, two cases of incoming packets are considered. First, a worst-case scenario when only packets generating the biggest possible crossproducts are classified (which leads to the lowest throughput). Second, a mean-case scenario where the real network data traces are used.

For less complex rulesets (*fw1_05_05* and *fw2_05_05*), the increase in throughput is around 25 % for the worst-case scenario. As expected, a higher increase is seen for a more complex ruleset (*acl*). In this case, an increase in throughput is up to 76 %. Thanks to memory duplication, the increase in throughput can be traded for an increase in memory requirements.

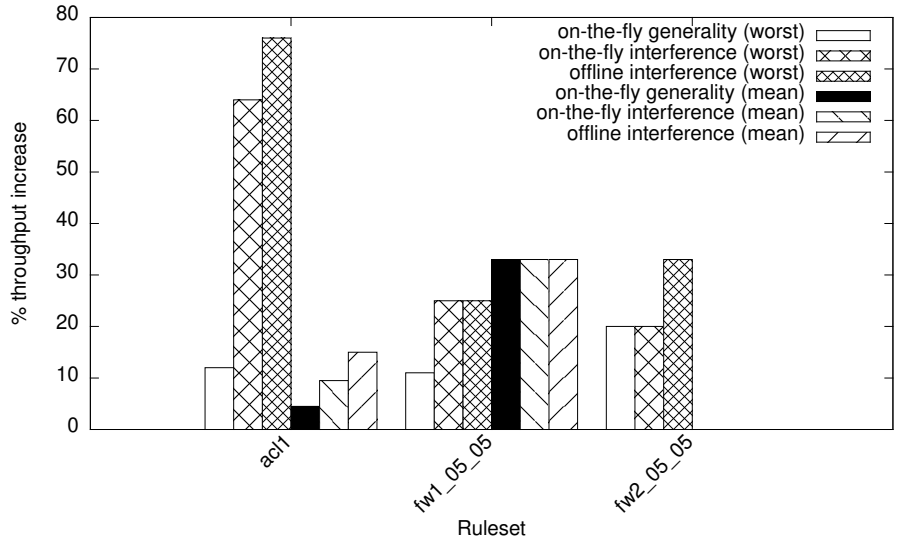

Figure 4.2: Throughput increase achieved by offloading rules for different configurations.

Exact Match Packet Classification

For exact match packet classification, the optimized Cuckoo hashing architecture [IP3][IP5] is used. Measurements of FPGA resource requirements for Xilinx technology are based on design implementations for the UltraScale+ XCVU9P chip using Vivado 2018.2 tool and for Intel are based on implementation for the Stratix10 1SG280HU2F50E1VG chip using Quartus Prime 18.1 Pro. The architecture is able to achieve working frequency (F_{max}) of more than 400 MHz for every evaluated configuration on both chips. The results presented are for architecture with three hash functions, 32 b wide arbitrary data (action) and 296 b wide key (which is a key needed for IPv6 flow matching). Different configurations of the architecture are considered - different sizes of the tables (memory rows) and different number

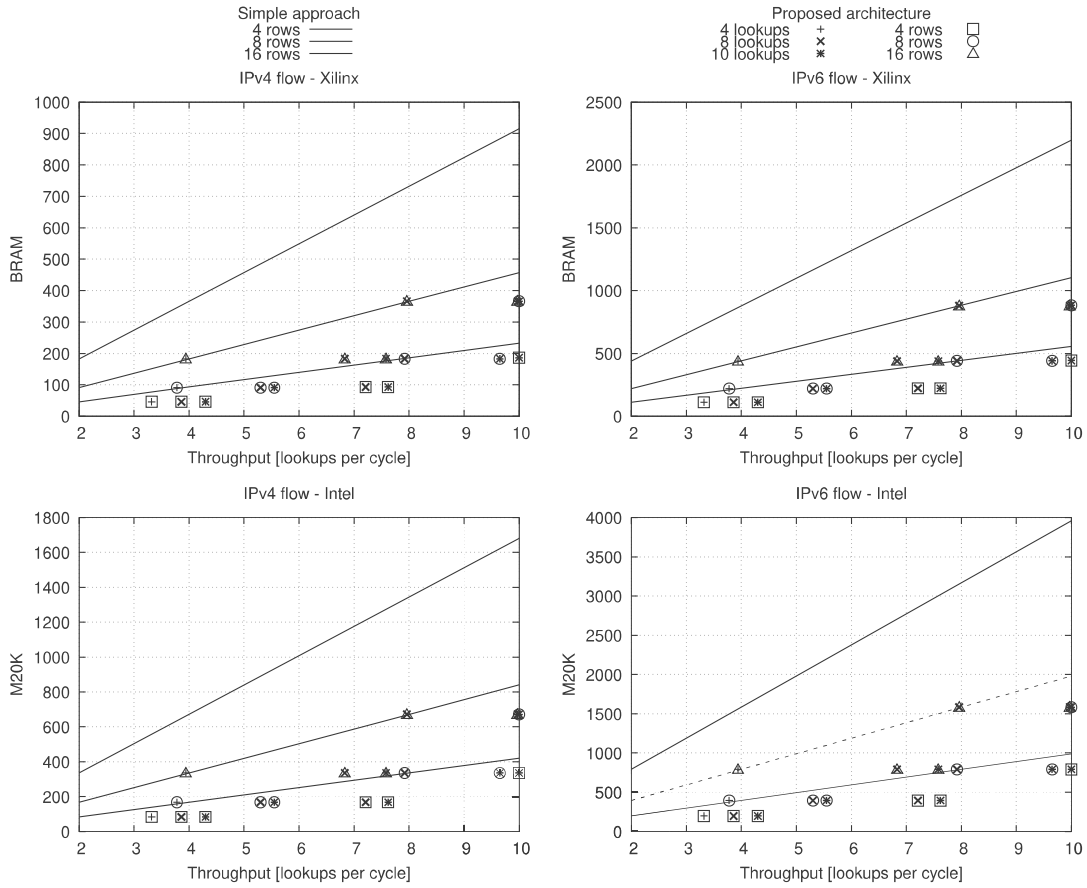

Figure 4.3: The relation between utilized memory and achieved throughput for different FPGAs and IPv6 flows when using three hash functions.

of parallel lookups that are attempted each clock cycle. Every distributed memory row in the presented results corresponds roughly to 2 765 effective rules that the table can hold.

Figure 4.3 captures the relations between lookups per cycle and memory tiles utilizations for the designed architecture. Results for a simple replication scheme (lines) form a baseline for evaluation of the designed optimization approach. With the Cuckoo hashing table using 16 rows (dotted line and triangle points), it is possible to achieve nearly twice the throughput without any memory duplication when using the proposed architecture with only four lookups (cross). If we use versions with 8 ('x') or 10 (star) lookups per cycle, the speedup is even further amplified and nearly 7 or 7.5 times higher throughput is achieved with no additional memory requirements. Additionally, when utilizing two replicas of memory, the proposed approach can achieve nearly the full throughput of 10 lookups per cycle. In other words, we achieve 99.7 % of throughput with only 40 % of used memory compared to simple replication.

A basic look at the achievable throughput under realistic deployment is provided in figure 4.4. It shows the achieved throughput for different numbers of block memory rows on captured network traces. As a reference, a throughput for packets with random uniformly distributed identifiers is shown. For specific network traces, two types of results are shown - results measured using packets truncated to 64B (packet window) and results measured using full packets (byte window). On both network traces, the architecture shows

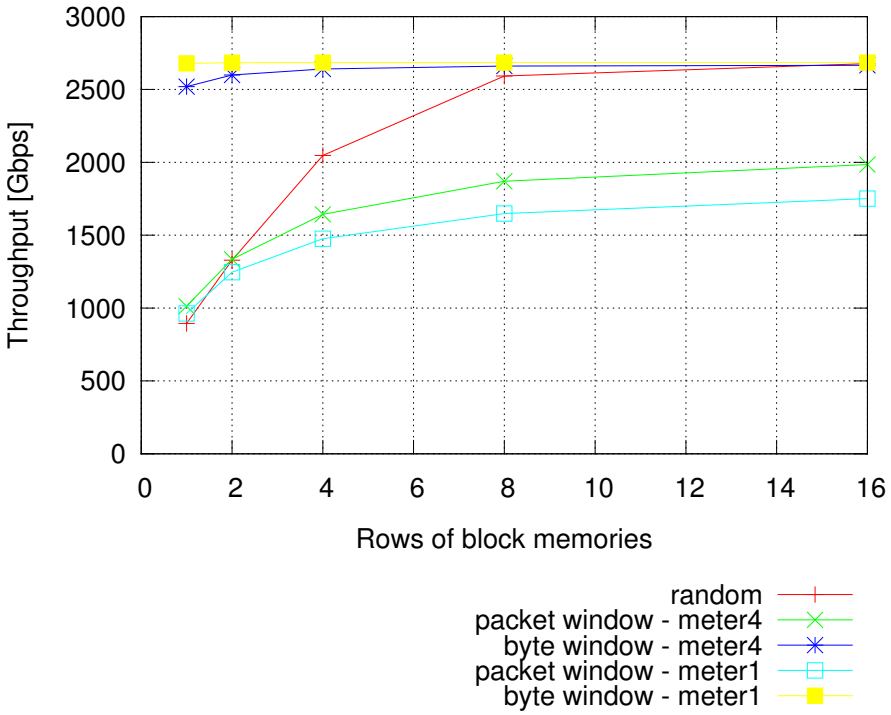

Figure 4.4: Throughput of the optimized Cuckoo hashing on real network traces for architecture with ten lookups and memory replication of two.

similar behaviour. For the worst-case scenario (packet window) the throughput is overall lower compared to random-case and the more realistic average-case (byte window). The interesting fact to notice about the worst-case scenario for real network traces is that its throughput falls behind random identifier distribution more and more with the rising number of block memory rows. The behaviour would suggest a more prevalent occurrence of access collisions than random. This makes sense since multiple packets from the same flows (representing the same end-to-end communication) tend to arrive within small time windows. However, if we take into account realistic packet lengths (byte window), the observed decrease in throughput is far outweighed by the lower arrival rate of matching requests into the architecture.

Due to the favourable scaling, the architecture can achieve an unprecedented throughput of 2.4 Tbps with an effective capacity of over 44 000 IPv4 5-tuple (flows) rules using on-chip block memories for the cost of only 366 BlockRAM tiles on Xilinx FPGAs [IP4] or 672 M20K tiles on Intel FPGAs [IP5].

External Memory Cache

External memories can be used to increase the rule capacity of packet classification. While the external memories offer an increased number of entries, the pattern of accessing them is different and might introduce unwanted stalls. To remove this downside, a cache architecture was proposed [IP5]. Measurements are based on design implementations for the UltraScale+ XCVU7P chip using Vivado 2019.1 tool.

The proposed approach was compared to three other possible implementations designed to fulfil the same functionality. First of all, a trivial implementation using sequential au-

tomaton. It issues a memory transaction for each requested operation and waits for the result to update it and write it back. The second approach is a trivial cache using the same sequential automaton extended with a traditional memory cache. The final approach is a trivial pipelined design with sufficient stages to cover the memory latency. However, it stalls when a collision occurs (2 accesses to the same memory address) to prevent data hazards.

The comparison of the architectures with 16 cache entries in table 4.3 consists of two results. First is the throughput (based on clock cycles needed for processing the whole input dataset). The value is shown as a ratio between the actual throughput and a maximal theoretical limit of the throughput (throughput if the stalls would never happen). A higher throughput value means faster processing and fewer or shorter stalls. The second value is the number of external memory transactions needed for processing the whole input dataset. The value is also represented as a ratio between the actual number of external memory transactions and a maximal theoretical limit (each request is a cache miss and accesses the external memory). In this case, a lower value means fewer external memory transactions are needed.

Table 4.3: Comparison of different cache architectures.

Memory access pattern	Trivial	Trivial cache	Pipeline	Proposed
random	0.125-1.000	0.138-0.936	0.501-1.000	0.500-1.000
dst IP hashing	0.125-1.000	0.391-0.546	0.221-1.000	1.000-0.432
src and dst IP hashing	0.125-1.000	0.358-0.565	0.221-1.000	1.000-0.433
full flow classification	0.125-1.000	0.342-0.576	0.223-1.000	1.000-0.438

The proposed architecture outperforms all of the other implementations and reaches almost the theoretical limits for almost every dataset, which translates to full throughput for a given frequency. Only the results for the random dataset are close to equal with the trivial pipeline because the proposed architecture uses a limited transaction table. When the transaction table is saturated, the design begins to stall and is downgraded to the trivial pipeline implementation. For real network traces, the architecture outperforms every other implementation at least by a factor of 3 when it comes to throughput. Another benefit of the architecture is that the number of memory transactions is reduced by a factor of 2.

Basic resource utilization of the architecture for different numbers of cache entries is shown in Table 4.4. More cache entries means that the probability of cache miss is lower. The configuration that was used worked with the 64b wide memory words. As expected, the resource utilization goes up pretty linearly and is still manageable even for tens of cache entries. However, the maximal frequency slowly decreases. The results show that the architecture with 16 or 32 entries should be sufficient to satisfy at least the frequency of 200 MHz, which means possibly up to 200 Mp/s (or in this case, 200 million operations carried out). This is more than enough for packet processing at the line rate of at least 100 Gb/s.

4.2 Conclusions

The results show that it is possible to achieve high throughput (100 Gbps and more) using new hardware architectures. The architectures are flexible enough and can be configured

Table 4.4: Resource utilization and frequency of the proposed architecture.

Size	LUTs	Registers	Frequency
4	2849	1439	>350 MHz
8	4151	2484	>350 MHz
16	7862	4573	285 MHz
32	16241	8750	225 MHz
64	31491	17103	188 MHz

based on the P4 description. High flexibility is therefore achieved as well. It can be concluded that the thesis confirms the initial hypothesis formulated in 1.1. This enables efficient P4-programmable packet classification on FPGA. All defined research objectives have been successfully completed.

The main contributions of the thesis are new flexible and optimized architectures for packet classification on FPGA that have these key attributes:

- **High flexibility**. The architectures are highly configurable. All standard types of matches are supported, as well as different capacities. This enables the packet classification to be used for a wide array of use cases. Also, the architectures can be used for mapping P4 to the FPGA.
- **High performance**. Wire-speed throughput for 100 Gbps links and faster can be feasibly achieved.
- **Memory efficiency**. The approaches use memory efficiently without unnecessary redundancies. This allows them to be used in more complex packet processing designs.
- **Scalability**. The architectures can be scaled to support more dimensions and higher throughputs. This allows to adjust throughput and required hardware resources. Additionally, a caching mechanism allows fast updates of memory items, which allows the classification to be scaled to large rule capacities by efficient usage of external memories.

The research conducted as part of the thesis also introduces these additional related contributions:

- Hardware architecture capable of high-speed encryption and authentication of network traffic using IPsec protocol. The architecture contains efficient hardware implementation of AES (Advanced Encryption Standard) encryption algorithm and the GCM (Galois Counter Mode) mode of operation.
- Hardware implementation of general configurable packet parser. The implementation allows efficient extraction of packet header fields used by the packet classification.

4.3 Deployment and Future Work

Parts of the presented research were deployed within the Netcope P4 compiler [37]. The Netcope P4 is a compiler that translates the P4 description into FPGA design. It was

developed in the scope of *NFV200* research project [3] in cooperation with the Cesnet association and supported both Xilinx and Intel FPGAs. The compiler was later used to generate the base infrastructure for the IPsec cryptography architecture described in [P2]. The technology and the compiler were acquired by Intel in 2020. Intel also participates in the P4 ecosystem with Intel Tofino, a P4-programmable Ethernet switch ASIC. The packet classification research mentioned in the thesis was also used in probes used by Police of the Czech Republic to defend against cyber criminality and in probes used to defend the CESNET network perimeter. The external memory cache architecture was used to accelerate virtual Open vSwitch in the scope of *OVS* research project [1].

Computer networks are still evolving further, and 400 Gbps and 1 Tbps Ethernet will soon become the norm. There is still a need for further optimizations and new packet classification architectures. More specialized architectures might be leveraged for specific use cases. There is still more room for improvement in optimizing the P4 description itself. Some basic ideas of merging tables and sharing resources between tables were presented in this thesis, but the presented approaches can be further improved.

Bibliography

[1] *Acceleration platform for virtual switches* [https://starfos.tacr.cz/en/projekty/TH04010193]. Accessed: 2023-10-19.

[2] *P4 Reference Compiler*. GitHub. Available at: https://github.com/p4lang/p4c.

[3] *Platform for Acceleration of Network Functions Virtualization* [https://starfos.tacr.cz/en/projekty/TH02010214]. Accessed: 2023-10-19.

[4] *OpenFlow Switch Specification v1.4.0*. 2013.

[5] Barish, G. and Obraczka, K. World Wide Web caching: trends and techniques. *IEEE Communications Magazine*. 2000, vol. 38, no. 5, p. 178–184. DOI: 10.1109/35.841844.

[6] Benáček, P., Puš, V. and 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, p. 148–155.

[7] Benáček, P., Puš, V., Kubátová, H. and Čejka, T. P4-to-VHDL: Automatic generation of high-speed input and output network blocks. *Microprocessors and Microsystems*. 2018, vol. 56, p. 22 – 33. ISSN 0141-9331.

[8] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N. et al. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* New York, NY, USA: ACM. july 2014, vol. 44, no. 3, p. 87–95. ISSN 0146-4833.

[9] Cabal, J., Benáček, P., Kekely, L., Kekely, M., Puš, V. et al. Configurable FPGA Packet Parser for Terabit Networks with Guaranteed Wire-Speed Throughput. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, 2018, p. 249–258. DOI: 10.1145/3174243.3174250. ISBN 978-1-4503-5614-5. Available at: https://www.fit.vut.cz/research/publication/11674.

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

[11] Degermark, M., Brodnik, A., Carlsson, S. and Pink, S. Small forwarding tables for fast routing lookups. In: *ACM Sigcomm*. 1997, p. 3–14.

[12] Dharmapurikar, S., Song, H., Turner, J. and Lockwood, J. 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, p. 61–70. ISBN 1-59593-580-0.

[13] Eatherton, W., Varghese, G. and Dittia, Z. *Tree bitmap: hardware/software ip lookups with incremental updates*. 2004. Available at: http://www.cs.cornell.edu/courses/cs419/2005sp/tree-bitmap.pdf.

[14] Gupta, P. and McKeown, N. *Algorithms for Packet Classification*. Available at: http://yuba.stanford.edu/~nickm/papers/classification_tutorial_01.pdf.

[15] Gupta, P. and McKeown, N. *Packet Classification on Multiple Fields*. Available at: http://yuba.stanford.edu/~nickm/papers/Sigcomm99.pdf.

[16] Gupta, P. and McKeown, N. Algorithms for packet classification. *IEEE Network*. 2001, vol. 15, no. 2, p. 24–32. DOI: 10.1109/65.912717.

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

[18] Ielmini, D. and Wong, H.-S. P. In-memory computing with resistive switching devices. *Nature Electronics*. 2018, vol. 1, p. 333–343. Available at: https://api.semanticscholar.org/CorpusID:57248729.

[19] Jain, S., Ranjan, A., Roy, K. and Raghunathan, A. Computing in Memory With Spin-Transfer Torque Magnetic RAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 2018, vol. 26, no. 3, p. 470–483. DOI: 10.1109/TVLSI.2017.2776954.

[20] Jiang, W. and Prasanna, V. K. Scalable Packet Classification on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. 2012, vol. 20.

[21] Kekely, L., Kucera, J., Pus, V., Korenek, J. and Vasilakos, A. V. Software Defined Monitoring of Application Protocols. *IEEE Trans. Comput.* Washington, DC, USA: IEEE Computer Society. february 2016, vol. 65, no. 2, p. 615–626. ISSN 0018-9340.

[22] Kekely, L., Žádník, M., Matoušek, J. and Kořenek, J. Fast Lookup for Dynamic Packet Filtering in FPGA. In: *17th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. Warszaw, Poland: IEEE Computer Society, 2014, p. 219–222. ISBN: 978-1-4799-4558-0.

[23] Kekely, M., Hynek, K. and Čejka, T. Pipelined ALU for effective external memory access in FPGA. In: *2020 23RD EUROMICRO CONFERENCE ON DIGITAL SYSTEM DESIGN (DSD 2020)*. Institute of Electrical and Electronics Engineers, 2020, p. 97–100. DOI: 10.1109/DSD51259.2020.00026. ISBN 978-1-7281-9535-3. Available at: https://www.fit.vut.cz/research/publication/12450.

[24] Kekely, M., Kekely, L. and Kořenek, J. Memory Aware Packet Matching Architecture for High-Speed Networks. In: *Proceedings of the 21st Euromicro*

Conference on Digital Systems Design. IEEE Computer Society, 2018. DOI: 10.1109/DSD.2018.00017. ISBN 978-1-5386-7376-8. Available at: https://www.fit.vut.cz/research/publication/11819.

[25] Kekely, M., Kekely, L. and Kořenek, J. General memory efficient packet matching FPGA architecture for future high-speed networks. *Microprocessors and Microsystems*. Elsevier Science. 2020, vol. 73, no. 3. DOI: 10.1016/j.micpro.2019.102950. ISSN 0141-9331. Available at: https://www.fit.vut.cz/research/publication/12138.

[26] Kekely, M. and Kořenek, J. Packet Classification with Limited Memory Resources. In: *In proceedings 2017 Euromicro Conference on Digital System Design*. Institute of Electrical and Electronics Engineers, 2017, p. 179–183. DOI: 10.1109/DSD.2017.61. ISBN 978-1-5386-2145-5. Available at: https://www.fit.vut.cz/research/publication/11550.

[27] Kekely, M. and Kořenek, J. Optimizing Packet Classification on FPGA. In: *PROCEEDINGS 2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. Institute of Electrical and Electronics Engineers, 2023, p. 7–12. ISBN 978-83-503-3276-7. Available at: https://www.fit.vut.cz/research/publication/12805.

[28] Kennedy, A., Wang, X., Liu, Z. and Liu, B. Low power architecture for high speed packet classification. In: *Proceedings of the 2008 ACM Symposium on Architecture for Networking and Communications*. 2008. ANCS '08.

[29] Kirsch, A., Mitzenmacher, M., Baohua, Y., Yibo, X. and Jun, L. *Using a Queue to De-amortize Cuckoo Hashing in Hardware*. 2007. Available at: http://www.eecs.harvard.edu/~michaelm/postscripts/aller2007.pdf.

[30] Kořenek, J. and Kekely, M. Mapping of P4 Match Action Tables to FPGA. In: *Preceedings of 27TH INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE LOGIC AND APPLICATIONS*. Institute of Electrical and Electronics Engineers, 2017. DOI: 10.23919/FPL.2017.8056768. ISBN 978-90-90-30428-1. Available at: https://www.fit.vut.cz/research/publication/11551.

[31] Kořenek, J., Puš, V. and 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, 2009, p. 165–166. Association for Computing Machinery. ISBN 978-1-60558-630-4.

[32] Lakshman, T. V. and Stiliadis, D. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *SIGCOMM Comput. Commun. Rev.* New York, NY, USA: ACM. 1998, vol. 28, no. 4, p. 203–214. ISSN 0146-4833.

[33] Le, H. and Prasanna, V. K. Scalable Tree-based Architectures for IPv4/v6 Lookup Using Prefix Partitioning. july 2012, vol. 61, no. 7, p. 1026–1039. ISSN 0018-9340.

[34] Lee, H., Jiang, W. and Prasanna, V. K. Scalable High-Throughput SRAM-Based Architecture for IP Lookup Using FPGA. In: *International Conference on Field Programmable Logic and Applications*. 2008.

[35] Luo, Y., Xiang, K. and Li, S. Acceleration of decision tree searching for IP traffic classification. In: *Proceedings of the 2008 ACM Symposium on Architecture for Networking and Communications*. 2008. ANCS '08.

[36] Martinásek, Z., Hajný, J., Smékal, D., Malina, L., Kekely, M. et al. 200 Gbps Hardware Accelerated Encryption System for FPGA Network Cards. In: *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 2018, p. 11–17. DOI: 10.1145/3266444.3266446. ISBN 978-1-4503-5996-2. Available at: https://www.fit.vut.cz/research/publication/12244.

[37] Netcope. *Netcope P4: User Guide*. 2020.

[38] Nilsson, S. and Karlsson, G. Fast address lookup for internet routers. In: *IEEE Broadband Communications*. 1998, p. 11–22.

[39] Pagh, R. and Rodler, F. F. Cuckoo Hashing. In: *Algorithms - ESA 2001*. Springer Berlin Heidelberg, 2001, vol. 2161, p. 121–133. Lecture Notes in Computer Science. ISBN 978-3-540-44676-7.

[40] Pus, V., Tobola, J., Kosar, V., Kastil, J. and Korenek, J. Netbench: Framework for Evaluation of Packet Processing Algorithms. In: *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*. 2011, p. 95–96. DOI: 10.1109/ANCS.2011.25.

[41] Puš, V. and 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.

[42] Qi, Y., Fong, J., Jiang, W., Xu, B., Li, J. et al. Multi-dimensional packet classification on FPGA: 100 Gbps and beyond. In: *2010 International Conference on Field-Programmable Technology*.

[43] Singh, S., Baboescu, F., Varghese, G. and Wang, J. Packet classification using multidimensional cutting. In: *Conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2003, p. 213–224. ISBN 1-58113-735-4.

[44] Song, H. and Lockwood, J. W. 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, 2005, p. 238–245. ISBN 1-59593-029-9.

[45] Srinivasan, V. and Varghese, G. Faster ip lookups using controlled prefix expansion. In: *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 1998, p. 1–10.

[46] Srinivasan, V., Varghese, G., Suri, S. and Waldvogel, M. Fast and scalable layer four switching. *SIGCOMM Comput. Commun. Rev.* New York, NY, USA: ACM. 1998, vol. 28, no. 4, p. 191–202. ISSN 0146-4833.

[47] Tank, G. P., Dixit, A., Vellanki, A. and Annapurna, D. *Software-Defined Networking: The New Norm for Networks*. April 2012. Available at: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

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

[49] Taylor, D. and Turner, J. ClassBench: a packet classification benchmark. In: *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies*. 2005, vol. 3, p. 2068–2079 vol. 3. DOI: 10.1109/INFCOM.2005.1498483.

[50] The P4 Language Consortium. *The P4 Language Specification: Version 1.0.5*. 2018.

[51] The P4 Language Consortium. *P4$_{16}$ Language Specification: Version 1.1.0*. 2018.

[52] Tobola, J. and Kořenek, J. Effective Hash-based IPv6 Longest Prefix Match. In: *IEEE Design and Diagnostics of Electronic Circuits and Systems DDECS'2011*. IEEE Computer Society, 2011, p. 325–328. ISBN 978-1-4244-9753-9. Available at: http://www.fit.vutbr.cz/research/view_pub.php?id=9602.

[53] Wang, H., Sun, T. and Yang, Q. Minimizing area cost of on-chip cache memories by caching address tags. *IEEE Transactions on Computers*. 1997, vol. 46, no. 11, p. 1187–1201. DOI: 10.1109/12.644293.

[54] Yaxuan, Q., Lianghong, X., Baohua, Y., Yibo, X. and Jun, L. Packet Classification Algorithm: From Theory to Practice. In: *IEEE INFOCOM 2009 proceedings*. IEEE Communications Society, 2009. Available at: http://users.ece.cmu.edu/~lianghon/docs/infocom09-hypersplit.pdf.

Appendix A

Included Papers

A.1 Paper 1

Mapping of P4 Match Action Tables to FPGA

Mapping of P4 Match Action Tables to FPGA

Michal Kekely
Faculty of Information Technology
Brno University of Technology
Brno, Czech republic
Email: ikekelym@fit.vutbr.cz

Jan Korenek
Faculty of Information Technology
Brno University of Technology
Brno, Czech republic
Email: korenek@fit.vutbr.cz

***Abstract*—Current networks are changing very fast. Network administrators need more flexible and powerful tools to be able to support new protocols or services very fast. The P4 language provides new level of abstraction for flexible packet processing. Therefore, we have designed new architecture for memory efficient mapping of P4 match/action tables to FPGA. The architecture is based on DCFL algorithm and is able to balance the processing speed and available memory resources.**

I. Introduction

In recent years, the capacity of network links has grown steadily. Network traffic processing needs flexible and faster algorithms. The time to process one packet is shortened. For current 100 Gbps networks it is necessary to process one packet in 5 ns, for the 400 Gbps networks it will be only 1.25 ns. In order to achieve high-speed network traffic processing, it is necessary to use an appropriate hardware acceleration method.

Moreover network administrators need more flexible tools. Many different network devices from different vendors need to work within one network. These devices have to be configured and since they can have different implementations and different target architectures this might be pretty time comsuming task. Bosshart *et al.* [1] suggested a flexible mechanism for describing packet processing. The P4 language has been specified at Stanford University to enable high-level definition of packet processing. The main goals of the language are to be reconfigurable, protocol independent and target independent. Therefore a network administrator does not have to understand the target technology. This is especially beneficial when ASICs and FPGAs are used. Last but not least adding support for new protocols simply means slightly altering the P4 program.

Packet classification is one of the crucial functionalities that is often carried out in HW. Therefore, we have analysed properties of the DCFL algorithm [4] to design new hardware architecture, which is highly memory efficient and is able to scale the throughput even up to 100 Gbps at the cost of additional hardware resources.

The rest of the paper is structured as follows. Problem definition and objectives of PHD thesis are outlined in Section II. Current state of the work and some results are in Section III. The paper is concluded by mentioning plans for future work in Section IV.

II. Related work

P4 language [1] defines 3 main processing parts. First of all every packet needs to be parsed. Values of parsed header fields are then used in Match Action Tables (MAT) to determine actions applied to the packet. These actions can modify selected header fields, add or delete headers and drop or forward the packet. Finaly the packet needs to be assembled back together from the modified and possibly new headers.

We mainly focus on transforming Match Action Tables. MATs are part of the P4 language that represent packet classification. These tables can classify based on different header fields, each with different type of match applied (exact, ternary, prefixes or LPM). Additionaly each table can be applied only when certain conditions are met.

Different state-of-the-art approaches to packet classification have different advantages and disadvantages. Hash based approaches such as Cuckoo Hashing [9] can be utilized to implement classifications that use only exact matches. However, they are unable to deal with any other type of match. For longest prefix match, trie [10] structures seem to give the best results. Alternatively more general solutions like TCAMs need to be deployed. TCAMs can deal even with ternary type of match but take up way too much logic resources, which means their capacity is limited. Many other research papers suggest hardware architectures that use the geometric representation of the classification problem, where packet header fields are dimensions and the classification is the searching in the n-dimensional space. Algorithms such as HyperSplit [2] and decision forest [3] use pipelined hardware architecture, where finding the rule is split among multiple pipeline stages. However, considerable amount of memory resources has to be used.

Before designing the actual mapping of MATs to FPGA we first chose to focus on designing and implementing new flexible hardware architecture for classifying packets. The proposed design leverages different single dimensional approaches for exact and prefix based matching to create architecture that supports multiple dimensions and can be scaled well even for higher throughputs.

III. Architecture

Architecture we propose is flexible, supports multiple dimensions and can be used as the most general one that implements every table for which no better approch was

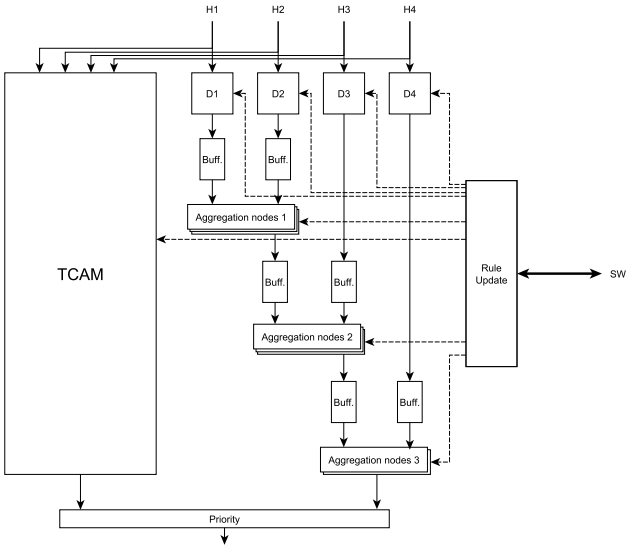

Fig. 1. Proposed architecture for DCFL with memory duplication and additional TCAM.

found. This mainly means big tables with multiple different dimensions and different types of matches.

We utilize DCFL algorithm [4] to design new scalable hardware architecture, which is able to balance processing speed and available hardware resources. Architecture, illustrated in Figure 1, works in a pipeline-like fashion, which leverages the fact that packet classification is split into single dimensions and series of aggregations of the results from those single dimensions. Memory requirements of aggregation nodes are extremely low since probabilistic approches such as Bloom filters [5] can be used.

Bottleneck of the original architecture are the aggregation nodes. To optimize the original DCFL architecture we propose two main techniques. First of which is memory duplication, the second being TCAM offload. Memory duplication exploits the fact that agreggation node have extremely low memory requirements. It is possible to simply duplicate memories used in aggregation nodes. By duplicating memories we gain the ability to essentially do more than one memory access per clock cycle. This can further be amplified by using dual port block memories. Obviously, to do more than one memory access per cycle the control logic also needs to be more complex, so the number of on-chip resources needed (LUTs, FFs ...) also increases. TCAM offload introduces small TCAM to the architecture. It is then possible to identify rules that are causing the DCFL algorithm to not perform efficiently and classify those rules in the TCAM instead.

Graph in figure 2, show scaling of block RAMs needed for increasing throughput of the architecture. It also shows comparison to other approaches (decision forest [3], BV-TCAM [6] and HyperCuts [8]). The architecture runs at 200 MHz and throughput was computed for worst-case scenario of shortest possible packets (64B). We can see that the number

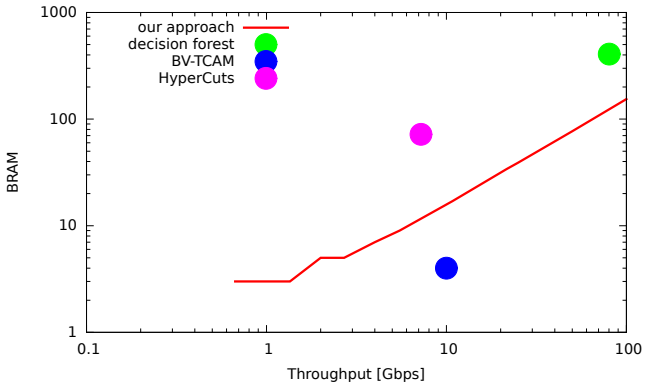

Fig. 2. Graph showing number of BRAMs needed and throughput of proposed architecture compared to other approaches.

of BRAMs scales linearly with the throughput. It also means that the memory requirements stay manageable even for higher throughputs. Note that the results shown for our architecture are for capacity of around 5500 rules, whereas results of other architectures may have different rule capacities.

IV. Conclusions and Future Works

The paper presented new scalable hardware architecture for packet classification based on DCFL algorithm. The proposed architecture is memory efficient and is able to balance hardware resource and processing speed. It means that networking applications and devices can fully utilize available resources to achieve maximal throughput. Moreover, the architecture is not limited in the number or types of supported dimensions.

In the future main focus will be around automatization of the actual mapping of P4 MATs to FPGA. A new compiler will be implemented. It should be able to decide which implementation is best to be used for MAT based on information that can be obtained from a P4 program.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese and D. Walker, *P4: Programming Protocol-Independent Packet Processors, CRC*, July 2014.
[2] Q. Yaxuan, X. Lianghong, Y. Baohua, X. Yibo and L. Ji, *Packet Classification Algorithms: From Theory to Practice*, In *Proc. IEEE INFOCOM*, 2009.
[3] W. Jiang and V. K. Prasanna, *Scalable Packet Classification on FPGA, IEEE Trans. on VLSI*, vol. 20, no. 9, 2012.
[4] D. E. Taylor and J. S. Turner, *Scalable Packet Classification using Distributed Crossproducting of Field Labels*, in *IEEE INFOCOM*, 2005.
[5] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, *Fast Packet Classification Using Bloom Filters,* in *Proc. ANCS*, 2006.
[6] H. Song and J. W. Lockwood, *Efficient packet classification for network intrusion detection using FPGA,* in *Proc. FPGA*, 2005.
[7] *OpenFlow Switch Specification,*, ver. 1.4.0, 2013.
[8] Y. Luo, K. Xiang, and S. Li, *Acceleration of decision tree searching for IP traffic classification,* in *Proc. ANCS*, 2008.
[9] R. Pagh and F. Rodler, *Cuckoo Hashing, Algorithms - ESA*, 2001.
[10] R. Briandais, *File searching using variable length keys,* in *Proc. Western J. Computer Conf*, 1959.

A.2 Paper 2

Packet Classification with Limited Memory Resources

2017 Euromicro Conference on Digital System Design

Packet Classification with Limited Memory Resources

Michal Kekely
Faculty of Information Technology
Brno University of Technology
Brno, Czech republic
Email: ikekelym@fit.vutbr.cz

Jan Korenek
Faculty of Information Technology
Brno University of Technology
Brno, Czech republic
Email: korenek@fit.vutbr.cz

***Abstract*—Network security and monitoring devices use packet classification to match packet header fields in a set of rules. Many hardware architectures have been designed to accelerate packet classification and achieve wire-speed throughput for 100 Gbps networks. The architectures are designed for high throughput even for the shortest packets. However, FPGA SoC and Intel Xeon with FPGA have limited resources for multiple accelerators. Usually, it is necessary to balance between available resources and the level of acceleration. Therefore, we have designed new hardware architecture for packet classification, which can balance between the processing speed and hardware resources. To achieve 10 Gbps average throughput the architecture need only 20 BlockRAMs for 5500 rules. Moreover, the architecture can scale the processing speed to wire-speed throughput on 100 Gbps line at the cost of additional memory resources.**

I. Introduction

In recent years, the capacity of network links has grown steadily. Network traffic processing need flexible and faster algorithms. The time to process one packet is shortened. For current 100 Gbps networks it is necessary to process one packet in 6.7 ns, for the 400 Gbps networks it will be only 1.6 ns. In order to achieve high-speed network traffic processing, it is necessary to use an appropriate hardware acceleration method.

Packet filters are widely used to ensure the security of computer networks. The filters are configured by a set of rules, which are defined on selected packet header fields. The packet filter finds a rule for every input packet and applies the appropriate action. Finding a rule for an input packet is defined by an operation called packet classification. Packet classification is used not only in packet filters but also in other security devices such as IDS/IPS systems, network traffic shaping or gathering statistical information. In P4 language [1], packet classification is the key operation to implement match/action tables.

Many research papers deals with the acceleration of the packet classification in order to achieve high throughput in tens or hundreds of gigabits. First algorithms and hardware architectures HiCuts [3] and HyperCuts [2] use the geometric representation of the classification problem, where packet header fields are dimensions and the classification is the searching in the n-dimensional space. Many other algorithms such as HyperSplit [4] and decision forest [5] use pipelined hardware architecture, where finding the rule is split among multiple pipeline stages. The ruleset is usually transformed to the tree, which is split to multiple pipeline stages and stored in the on-chip memory. It is thus possible to achieve high throughput. However, the whole ruleset is stored in the on-chip memory and considerable amount of memory resources has to be used.

Effective utilization of the on-chip memory was demonstrated by the DCFL [6] algorithm, but only at the cost of time complexity, which can be significantly increased for some types of rules. the original DCFL algorithm doesn't provide any idea how to deal with these types of rules to increase processing speed.

The new Xilinx Zynq and Intel Xeon with FPGA provide wide scope for the acceleration of network functions. Usually, networking applications and devices need to accelerate at the same time multiple operations (packet classification, pattern matching, encryption, etc.). It means that it is not possible to allocate the entire FPGA only for one function. Instead, every function has limited hardware resources and has to adjust the throughput to available resources. However, current hardware architectures are primarily trying to achieve high throughput without possibility to scale throughput with respect to available on-chip resources.

Therefore we have analysed properties of the DCFL algorithm to design new hardware architecture, which is highly memory efficient and is able to scale the throughput at the cost of additional hardware resources. As can be seen in the results, the architecture utilizes only 20 BlockRAMs to store 5500 rules to achieve 10 Gbps average throughput. Moreover, we were able to scale the performance of proposed hardware architecture up to the wire-speed 100 Gbps throughput at the cost of reasonable hardware resources.

The rest of the paper is structured as follows. Related work is surveyed in Section II. We propose the classification architecture in Section III and the evaluation and experimental results are in Section IV. the paper is concluded in Section V.

II. Related work

Current approaches to packet classification focus mainly on throughput and number of rules supported. To achieve

978-1-5386-2146-2/17 $31.00 © 2017 IEEE
DOI 10.1109/DSD.2017.61

maximal throughput and high number of rules these algorithms sacrifice flexibility in terms of supporting different number of dimensions and different types of matches in each of the dimensions. Moreover the architectures used often use great amount of resources with limited options of scaling it down.

Packet classification based on bit-parallelism (or bit vectors, BV), proposed by Lakshman *et al.* [9], is a practical implementation that leverages the fact that rule updates are infrequent compared to search operations. The algorithm works in two stages. In the first stage parallel searches are carried out within each of the dimensions, resulting in bit vectors. Each bit of the vectors corresponds to one rule of classification, therefore their width is given by the number of rules used for classification. A bit is set if corresponding rule is matched in given dimension and is reset otherwise. After first stage every bit vector represents the set of all the rules matched in one dimension. Then the second stage has to find intersection of the sets matched within single dimensions. Since these sets are represented as bit vectors finding the intersection is reduced to bitwise AND operation among the bit vectors. The main problem of this approach is the width of bit vectors which increases with number of rules. Song *et al.* [10] presented architecture that combines bit vector approach with TCAMs. The architecture uses TCAMs for lookups within dimensions that require exact or prefix matches and tree-bitmap implementation of BV algorithm for source and destination port lookups. This architecture is optimized for classification based on network flow 5-tuples (source IP address, destination IP address, source port, destination port and L4 protocol) and therefore is not very flexible and was not shown to have the ability to scale to support different header fields.

With OpenFlow and other new emerging standards there is an increasing need to support more than the standard 5 dimensions. First specification of OpenFlow supports up to 12 dimensions with newer 1.4.0 version [11] supporting 42 different header fields. Therefore algorithms need to be able to scale to more dimensions, that might have different characteristics. Several of the possible approaches to support multiple dimensions are described in [12]. Grid of tries extends standard trie structure to two dimensions, however it is not easily extendable to more than two. General solution using crossproducting is more promising, but with no further optimizations uses up way too much memory and resulting cross-products are quite big. Other trie-based algorithms scale poorly with increasing number of dimensions. Additionally these algorithms need great amounts of memory and cannot be easily scaled to higher throughputs. McKeown *et al.* [13] proposed using recursive flow classification (RFC). They suggest that packet classification can be viewed as mapping of N bits (given by the header fields) to M bits representing rule or action matching given packet. Obviously M is expected to be way lower than N. Directly implementing such a mapping would require $2\hat{N}$ entries in memory, therefore RFC algorithm splits this mapping into multiple stages that recursively map one set of values to a smaller set of values. Once again the downfall of this approach is memory needed. Especially when the number of phases is low the memory requirements are very high.

Different approaches try to utilize architectures based on building decision trees. Many of those algorithms were not designed with FPGA implementation in mind, however some of them can be bent to run efficiently on FPGAs. HiCuts [3] and HyperCuts [2] are examples of such algorithms. The main idea is to progressively cut the whole searched space represented by classification dimensions into small enough parts (usually representing 1 or several rules). Different heuristics can be used to decide how to cut the space. Resulting trees tend to have many nodes. Additionally adding or removing rules leads to need of rebuilding the whole tree. A way to increase throughput of HyperCuts was introduced by Luo *et al.* [14]. Their method called explicit range search uses new methods to cut ranges in dimensions and then search within the ranges. This leads to increased throughput for the price of needing to store explicit marks in memory. Kennedy *et al.* [15] implemented simplified version of HyperCuts algorithm with the goal of reducing power consumption and increase power efficiency. They were able to lower the frequency to only 32 MHz which however means a throughput of only 0.47 Gbps.

Prasanna *et al.* [5] pushed the idea of building decision tree even further. They have observed that HyperCuts and similar decision-tree-based algorithms do not efficiently deal with rules that have too much overlap with each other. In such cases many rules need to be duplicated and the resulting tree (and required memory) can explode exponentially with number of dimensions. To combat this a decision forest is introduced. Ruleset is split into subsets and smaller decision trees are built for each subset. Rules within the subset are chosen so that they have as little overlap as possible and they specify nearly the same dimensions. Additionally two other techniques are used to optimize HyperCuts algorithm. Rule overlap reduction stores rules that should be replicated in a list in each internal node instead of actually replicating it into all the child nodes. Precise range cutting is used to determine cutting points which will result in the least number of rule duplications instead of deciding number of cuts for a field.

Taylor *et al.* [6] introduced Distributed Crossproducting of Field Labels (DCFL). This algorithm decomposes classification into single dimensions and can be easily parallelized. Moreover it uses Bloom Filters [7] and labeling technique which leads to low memory and logic requirements. Because of this several parts of the architecture can be duplicated to increase throughput while still maintaining reasonable usage of on-chip memories and logic.

III. Architecture

Our goal was to design an architecture that would be flexible. The architecture should scale well both with increasing number of dimensions as well as with required throughput. Because of this we needed an approach that starts with low

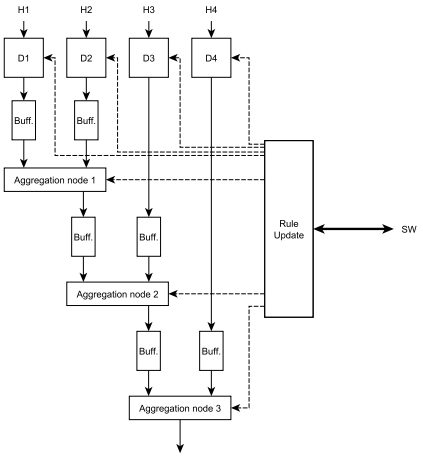

Fig. 1. Top level architecture of original DCFL algorithm.

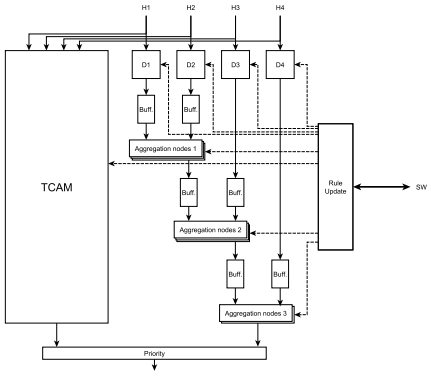

Fig. 2. Proposed architecture for DCFL with memory duplication and additional TCAM.

memory requirements and low throughput and can also be scaled up.

We utilize DCFL algorithm to design new scalable hardware architecture, which is able to balance processing speed and available hardware resources. Original DCFL based architecture is illustrated for 4-dimensional classification by figure 1. Architecture works in a pipeline-like fashion, which leverages the fact that packet classification is split into single dimensions and series of aggregations of the results from those single dimensions. First stage of pipeline consists of blocks *D1*, *D2*, *D3* and *D4*. These are responsible for classifying incoming packets within first, second, third and fourth dimension of classification respectively. Implementation of these blocks is dependant on the type of matching that needs to be done in each of the dimensions. This could mean using cuckoo-hash based approach [16] in case of exact match or trie based solutions [17] and TCAMs for matching prefixes. Additionally each dimension can be classified independently.

Results of single dimension classifications are stored in distributed memory buffers. Main function of these buffers is to act as synchronization between different stages of the pipeline as the number of cycles needed for different blocks of the architecture to process a single packet may vary based on matched rules. Capacity of the buffers does not need to be large, because the number of results that can match one packet in a single dimension is with high probability lower than 5 (as shown in [6]). To save even more distributed memory every unique value in each dimension is represented by unique label. This label has lower bit width than the value it represents. For example, if we know that the number of unique IPv4 addresses (or prefixes) is lower than 1024, we can use 10 bit wide labels instead of working with 32 bit wide IPv4 address values.

Every other pipeline stage aggregates two sets of results *R1* and *R2* from previous stages into one new result set *R*. First of all cross-product *R1xR2* of the two input sets is computed. Afterwards the new result set *R* needs to be found such that *R* is subset of the cross-product *R1xR2* and every element of *R* is part of some rule from ruleset of classification. Bloom filter arrays and Meta-label indexing are used to represent sets of every element that is part of some rule from ruleset of classification. Finding set *R* then simply means deciding set membership for every element of *R1xR2*.

Bottleneck of the architecture are the aggregation nodes. Each set membership query needs at least one memory access. Even though number of unique prefixes in each dimension, that match a packet, are quite low the resulting cross-product can in worst case consist of around 40 or even more elements. This means that to classify a single packet in worst-case scenario we need 40 memory accesses, which means 40 clock cycles. The throughput can be therefore severely impacted, especially on short packets.

To optimize the original DCFL architecture we propose two main techniques. First of which is memory duplication, the second being TCAM offload. Resulting architecture is illustrated by figure 2.

Memory duplication exploits the fact that using labels to represent unique prefixes (or values) in individual dimensions with addition of using Bloom filter arrays leads to extremely low memory requirements. It is possible to simply duplicate memories used in aggregation nodes. By duplicating memories we gain the ability to essentially do more than one memory access per clock cycle. This can further be amplified by using dual port block memories. It is also possible to duplicate some memories multiple times increasing memory requirements and throughput even more. Obviously, to do more than one memory access per cycle the control logic also needs to be more complex, so the number of on chip resources needed (LUTs, FFs ...) also increases. All of this allows us to tailor architectures throughput and memory requirements to our needs, which can be especially useful when we need to guarantee certain throughput while maintaining memory requirements as low as possible or when we want to get highest possible throughput with limited memory resources.

The technique of memory duplication can be most efficiently used when the actual ruleset of the classification or at least its core is known beforehand (in time of generating the architecture). Analyzing the ruleset can determine which aggregation nodes are critical (meaning they have to process cross-products of biggest sizes therefore need most cycles per packet). Memories within critical aggregation nodes can then be duplicated more times than memories within other aggregation nodes. Generally, lets assume that aggregation node A1 has to in worst-case process cross-product of size *S1* while aggregation node *A2* has to in worst-case process a cross-product of size *S2*. Memories of *A1* should be duplicated *(S1/S2)*-times as much as memories of *A2* to achieve the best possible trade-off between throughput and memory requirements.

Analysis of ruleset can not only help determine how much duplication should occur in each stage of DCFL but can also serve to identify rules that are interfering mostly with others. These are the rules that are part of the worst case matches (meaning they have a packet that matches them along with maximal number of other rules). Once we identified such rules we can offload them into parallel TCAM - hence the name TCAM offloading. To identify those rules a simple algorithm can be used. Let us assume we want to optimize worst case scenario for first aggregation node. To do so we want to decrease the maximal number of prefixes that can match a packet in either the first or the second (or both) dimension. We can build one dimensional tries for first and second dimension. While building those tries we also compute and store some additional information.

First of all for each node v we store a list $R(v)$ of all the rules corresponding to the prefix that this node represents. For example if we have rules *R1=(192.168.*.*, 10.10.10.*)* and *R2=(192.168.*.*, 192.168.1.1)*, both rules need to be stored at the node representing *192.168.*.** prefix in the first dimension. Secondly for each node v we compute $I(v)$ - the maximal number of relevant nodes (nodes which correspond to at least one rule) contained within any path that starts at root, goes through said node v and ends in any leaf and $G(v)$ - the number of relevant nodes on the path from root to node v (since we work with a tree there is only one such path). This numbers basically gives us how many other prefixes can be matched along with prefix represented by node v and number of more general prefixes. Next we pool nodes with the same G and I values together into pools. For each pool p we store also a new list $R(p)$ which is union of all the $R(v)$ lists, where v is any node from pool p. Finally to decrease number of prefixes that can match a packet by one we simply have to choose any pool p with maximal value of I and remove all the rules $R(p)$. Then we can update all the values and do this again.

Figure 3 illustrates why we need also the G values. Bold circles represent nodes with non-empty R. This example shows a situation where the worst-case scenario are 7 nested prefixes. Additionally there are 2 parallel branches both with 7 nested prefixes in them. Node x represents the most general prefix. We can reduce the worst-case nesting (maximal I value) by removing all the rules associated with this node ($R(x)$). This however might not be possible as the number of removed rules might be higher than capacity of TCAM. Alternatively we can remove all the rules associated with some of the other nodes representing nested prefixes. In the illustrated case this means removing both $R(y)$ and $R(z)$. If we removed only $R(y)$ the $I(x)$ and $I(z)$ values would not change (as they are in a parallel branch). This whole algorithm can be extended to also work with hierarchical tries for the aggregation nodes beyond the first one. Once again to utilize this technique fully a ruleset should be known before classification starts.

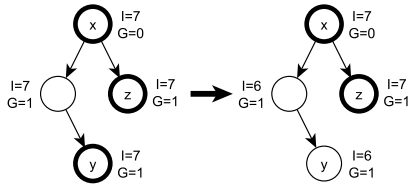

Fig. 3. Example of part of the trie built while analyzing ruleset.

TABLE I
CHARECTERISTICS OF RULESETS USED.

Name	Dimensions	Number of rules	Overlap
acl1	4	2406	high
fw1_05_05	5	733	medium
fw2_05_05	4	941	low

IV. RESULTS

In order to analyze how effective our solution is we implemented it using high-level synthesis. We used chip from Kintex-7 family of FPGAs. To demonstrate variety of rulesets we chose 3 sets of classification rules. These sets were generated by ClassBench [18] tool and are part of NetBench [19] framework. Table I shows the characteristics of the sets, main one being level of overlap between ranges or prefixes within rules. *acl1* is an example of ruleset with many overlapping ranges of port values, therefore many different rules can match the same packet within single dimension, which leads to DCFL not being very effective. *fw2_05_05* on the other hand has little overlap between rules and prefixes thus DCFL shows much better results. Finally *fw1_05_05* represents a middle ground.

Graphs in figures 4, 5 and 6 show scaling of block RAMs needed for increasing throughput of the architecture for rulesets *acl1*, *fw1_05_05* and *fw2_05_05*. The architecture runs at 200 MHz and throughput was computed for worst-case scenario of shortest possible packets (64B). We can see that the number of BRAMs scales linearly with the throughput. It also means that the memory requirements stay manageable even for higher throughputs. Note that the results shown for our architecture are for capacity of around 5500 rules, whereas results of other architectures may have different rule capacities (mainly decision forest has capacity of 10 000 rules). Additionally our architecture used quite small TCAMs

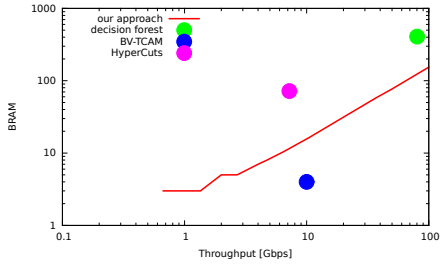

Fig. 4. Graph showing number of BRAMs needed and throughput of proposed architecture for ruleset *acl1* compared to other approaches.

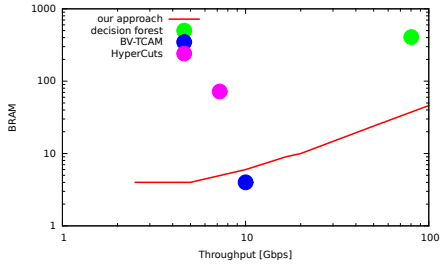

Fig. 5. Graph showing number of BRAMs needed and throughput of proposed architecture for ruleset *fw1_05_05* compared to other approaches.

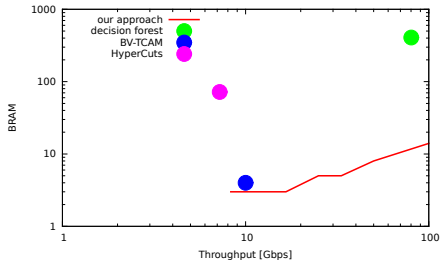

Fig. 6. Graph showing number of BRAMs needed and throughput of proposed architecture for ruleset *fw2_05_05* compared to other approaches.

(since number of unique values in single dimensions are a lot lower than number of rules) as engines for searching in single dimensions. We estimate that the number of BRAMs would be twice or thrice as big if other single-dimensional algorithms (Cuckoo Hash, Tries, etc.) were used. However even if we take into account previously mentioned information our approach still has around the same (or even better) memory requirements while having the ability to be scaled to various throughputs.

V. Conclusion

The paper presented new scalable hardware architecture for packet classification based on DCFL algorithm. The proposed architecture is memory efficient and is able to balance hardware resource and processing speed. It means that networking applications and devices can fully utilize available resources to achieve maximal throughput. Moreover, the architecture is not limited in the number or types of supported dimensions.

We introduced two techniques to lower the memory requirements and increase processing speed. High memory efficiency has been achieved. Our architecture can fit thousands of rules into memory available on current FPGA chips while maintaining high throughput. Furthermore the architecture scales very well and versions that use less memory and logic resources can be used if high throughput is not needed.

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese and D. Walker, *P4: Programming Protocol-Independent Packet Processors, CRC*, July 2014.

[2] S. Singh, F. Baboescu, G. Varghese and J. Wang, *Packet Classification Using Multidimensional Cutting,* in *Proc. SIGCOMMM*, 2003.

[3] P. Gupta and N. McKeown, *Classifying Packets with Hierarchical Intelligent Cuttings*, *IEEE Micro*, vol. 20, no. 1, 2000.

[4] Q. Yaxuan, X. Lianghong, Y. Baohua, X. Yibo and L. Ji, *Packet Classification Algorithms: From Theory to Practice*, In *Proc. IEEE INFOCOM*, 2009.

[5] W. Jiang and V. K. Prasanna, *Scalable Packet Classification on FPGA, IEEE Trans. on VLSI*, vol. 20, no. 9, 2012.

[6] D. E. Taylor and J. S. Turner, *Scalable Packet Classification using Distributed Crossproducting of Field Labels*, in *IEEE INFOCOM*, 2005.

[7] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, *Fast Packet Classification Using Bloom Filters,* in *Proc. ANCS*, 2006.

[8] V. Pus, *Fast and Scalable Packet Classification Using Perfect Hash Functions,* in *Proc. ACM/SIGDA International Synopsium on Field Programmable Gate Arrays*, 2009.

[9] T. V. Lakshman and D. Stiliadis, *High-speed policy-based packet forwarding using efficient multi-dimensional range matching,* in *Proc. SIGCOMM*, 1998.

[10] H. Song and J. W. Lockwood, *Efficient packet classification for network intrusion detection using FPGA,* in *Proc. FPGA*, 2005.

[11] *OpenFlow Switch Specification,*, ver. 1.4.0, 2013.

[12] V. Srinivasan, G. Varghese, S. Suri and M. Waldvogel, *Fast and Scalable Layer Four Switching,* in *Proc. SIGGCOM*, 1998.

[13] P. Gupta and N. McKeown, *Packet Classification on Multiple Fields*, in *Proc. SIGGCOM*, 1999.

[14] Y. Luo, K. Xiang, and S. Li, *Acceleration of decision tree searching for IP traffic classification,* in *Proc. ANCS*, 2008.

[15] A. Kennedy, X. Wang, Z. Liu, and B. Liu, *Low power architecture for high speed packet classification,* in *Proc. ANCS* 2008.

[16] R. Pagh and F. Rodler, *Cuckoo Hashing, Algorithms - ESA*, 2001.

[17] R. Briandais, *File searching using variable length keys,* in *Proc. Western J. Computer Conf*, 1959.

[18] D. E. Taylor and J. S. Turner, *ClassBench: a Packet Classification Benchmark,* in *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, 2007.

[19] Pus, V., Tobola, J. and Kosar, V. *Netbench: Framework for Evaluation of Packet Processing Algorithms*, *Symposium On Architecture For Networking And Communications Systems*, 2011.

A.3 Paper 3

Memory Aware Packet Matching Architecture for High-Speed Networks

2018 21st Euromicro Conference on Digital System Design

Memory Aware Packet Matching Architecture for High-Speed Networks

Michal Kekely
FIT BUT
Božetěchova 2, 612 66 Brno
Czech Republic
ikekelym@fit.vutbr.cz

Lukáš Kekely
CESNET, a. l. e.
Zikova 4, 160 00 Prague 6
Czech Republic
kekely@cesnet.cz

Jan Kořenek
IT4Innovations Centre of Excellence, FIT BUT
Božetěchova 2, 612 66 Brno
Czech Republic
korenek@fit.vutbr.cz

***Abstract*—Packet classification is a crucial operation for many different networking tasks ranging from switching or routing to monitoring and security devices like firewall or IDS. Generally, accelerated architectures implementing packet classification must be used to satisfy ever-growing demands of current high-speed networks. Furthermore, to keep up with the rising network throughputs, the accelerated architectures for FPGAs must be able to classify more than one packet in each clock cycle. This can be mainly achieved by utilization of multiple processing pipelines in parallel, what brings replication of FPGA logic and more importantly scarce on-chip memory resources.**

Therefore in this paper, we propose a novel parallel hardware architecture for hash-based exact match classification of multiple packets per clock cycle with reduced memory replication requirements. The basic idea is to leverage the fact that modern FPGAs offer hundreds of BlockRAM tiles that can be accessed (addressed) independently to maintain high throughput of matching even without fully replicated memory architecture. Our results show that the proposed approach can use memory very efficiently and scales exceptionally well with increased record capacities. For example, the designed architecture is able to achieve throughput of more than 2 Tbps (over 3 000 Mpps) with an effective capacity of more than 40 000 IPv4 flow records for the cost of only 366 BlockRAM tiles and around 57 000 LUTs.

I. Introduction

With the increasing capacity of network links, all network devices and systems need to speed up their packet processing. Current processors are not able to cope with network traffic even on 100 Gbps links. In order to achieve wire-speed processing with a throughput of 100 Gbps and more, network systems have to utilize FPGA or ASIC technology. The FPGA acceleration provides high performance and is highly configurable (flexible) as well. The flexibility is essential for any practical network system because traffic processing is changing with the introduction of every new protocol, application or service. Therefore, 40 Gbps network interface cards with FPGAs started to be deployed to data centers as hardware platforms for the acceleration [1] and will be probably more and more frequently used in the future.

Network traffic processing architecture can be easily described in the P4 high-level language [2] and then automatically mapped directly to an FPGA hardware accelerator [3], [4]. The P4 language has been designed at Stanford University in order to enable protocol, vendor and target independent definitions of packet processing. An integral part of the P4 language is the utilization of match/action tables as a basis to control processing of each input packet.

The match/action tables perform various forms of packet classification. During the classification, packets are matched against a set of rules, which are usually defined by values, ranges or prefixes of a few selected packet header fields. Generally, the classification is a mathematical problem of a multidimensional range search. Due to the ruleset size and complexity of rules, it is very difficult to perform matching at sufficient rate for wire-speed processing. Therefore, many hardware architectures have been designed to accelerate packet classification [5], [6], [7], [8], [9], [10], [11].

For 100 Gb network links, wire-speed throughput can be achieved only if a new packet is processed every 6.7 ns, which is only one clock cycle for 150 MHz clock. It means that multiple packets have to be processed within each clock cycle to achieve wire-speed 400 Gbps or 1 Tbps packet processing in FPGAs. Usually, the processing speed is increased by utilization of multiple parallel pipelines [12], [13], which require multi-port memories or memory replication. Unfortunately, both approaches significantly reduce throughput scalability at 400 Gbps or 1 Tbps speeds.

Therefore, we focus on the design of a new hardware acceleration technique for packet classification with efficient utilization of memory resources to achieve high-speed packet processing. We introduce novel hardware architecture that is able to scale the throughput of P4 match/action tables to more than 2 Tbps (over 3 000 Mpps) on current FPGAs while memory replication is significantly reduced compared to other approaches. The proposed concept is compared with simple pipeline/memory replication scheme and several possible optimizations are introduced.

II. Related Work

Currently, there are many different approaches to packet classification. Some of them focus on being as general as possible, supporting packet classification in multiple different dimensions and different types of match strength, such as range lookups, ternary matching or longest prefix match (LPM). However, the only way how to scale most of those approaches for higher throughputs is to utilize multiple copies of the same architecture operating in parallel.

978-1-5386-7377-5/18/$31.00 ©2018 IEEE
DOI 10.1109/DSD.2018.00017

Packet classification based on bit-parallelism (or bit vectors, BV), proposed by Lakshman et al. [14], is a practical implementation that leverages the fact that rule updates are infrequent compared to search operations. The algorithm works in two stages. In the first stage, parallel searches are carried out within each of the dimensions, resulting in bit vectors. Each bit of these vectors corresponds to one record in classification ruleset, therefore their width is given by the number of rules used. A bit is set to one if a corresponding rule is matched in given dimension and is reset to zero otherwise. After the first stage, every bit vector represents the set of all the rules matched in one dimension. Then the second stage has to find an intersection of the sets matched within single dimensions. Since these sets are represented as bit vectors finding the intersection is reduced to bitwise AND operation among the bit vectors. The main problem with this approach is the width of bit vectors which increases with the number of rules. Song et al. [15] presented architecture that combines bit vector approach with TCAMs. The architecture uses TCAMs for lookups within dimensions that require exact or prefix matches and tree-bitmap implementation of the BV algorithm for source and destination port lookups. This architecture is optimized for classification based on network flow 5-tuples (source IP address, destination IP address, source port, destination port and L4 protocol), therefore it is not very flexible and was not shown to have the ability to scale to support different header fields.

Several of different approaches to supporting multiple dimensions are described in [16]. A grid of Tries extends standard Trie structure to two dimensions however, it is not easily extendable to more than two. General solution using cross-products is more promising, but with no further optimizations uses up way too much memory and resulting cross-products are quite big. Other trie-based algorithms scale poorly with increasing number of dimensions. Additionally, these algorithms need great amounts of memory and cannot be easily scaled to higher throughputs.

Another group of approaches to classification tries to utilize architectures based on the construction of decision trees. Many of these algorithms are not designed with FPGA implementation in mind, however, some of them can be bent to be efficiently mapped into FPGA structure. HiCuts [6] and HyperCuts [7] are examples of such algorithms. The main idea is to progressively cut the whole searched space represented by classification dimensions into small enough parts (usually representing 1 or several rules). Different heuristics can be used to decide how to cut the space. But, resulting trees tend to have many nodes. Additionally adding or removing rules leads to the need of rebuilding the whole tree.

Prasanna et al. [17] pushed the idea of constructing decision trees even further. They have observed that HyperCuts and similar algorithms do not efficiently deal with rules that have too much overlap with each other. In such cases, many rules need to be duplicated and the resulting tree (and required memory) can explode exponentially with the number of dimensions. To combat this, a decision forest is introduced. Ruleset is split into subsets and smaller decision trees are built for each subset. Rules within each subset are chosen so that they have as little overlap as possible and that they specify nearly the same dimensions. Additionally, two other techniques are used to optimize HyperCuts algorithm. Rule overlap reduction stores rules that should be replicated in a list in each internal node instead of actually replicating it into all the child nodes. Precise range cutting is used to determine cutting points which will result in the least number of rule duplications instead of deciding number of cuts for a field.

Taylor et al. [5] introduced Distributed Crossproducting of Field Labels (DCFL). This algorithm decomposes classification into single dimensions and can be easily parallelized. Moreover, it uses Bloom Filters [9] and labeling technique to lower memory and logic requirements. The architecture was shown to be scalable even to higher throughputs [18], but only by using multiple copies of the memories. Because of this key features, the architecture can be duplicated to increase throughput while still maintaining reasonable usage of on-chip memories and logic. This idea was pushed further to build scalable architecture through memory duplication in [18].

In many cases, exact match packet classification is sufficient. This is prevalent mainly when IP flows are concerned. Effective approaches to exact match packet classification are usually based on hash tables. A sophisticated way of implementing hash tables is cuckoo hashing principle [19]. The main idea of cuckoo hashing is to increase the efficiency of memory utilization in the hash table by multiple parallel hash functions/tables. Each table uses one of the different hash functions for indexing. This means that if a new element cannot be inserted into the first table because of a conflict with an already existing item, it can still be inserted into one of the other tables through a different hash function. Even when the element cannot be inserted into any of the tables it can still be inserted by force, pushing out one of the previous occupants. The previous occupant can then be reinserted into the tables in the same manner. Using more tables and reinsertions allows the cuckoo hashing to keep the high lookup speed while decreasing the number of unresolvable conflicts and therefore increasing the effective capacity.

The cuckoo hashing approach is well suited for hardware because each hash table can work in parallel [20], [21]. These published implementations offer throughputs up to around only 100 Gbps, while in this paper we aim at achieving over 1 Tbps. Cuckoo hashing based packet classification is also effectively used to monitor or analyze network traffic in the idea of Software Defined Monitoring (SDM) [22]. Here, an external memory is utilized and achieved throughput is again shown to be sufficient only for up to 100 Gbps.

III. Architecture

Our main goal is to design an architecture for exact match packet classification that can accommodate high throughputs of multiple terabits per second. One way to achieve this would be to increase the clock frequency of basic cuckoo hashing architecture. This is possible to do only until a certain

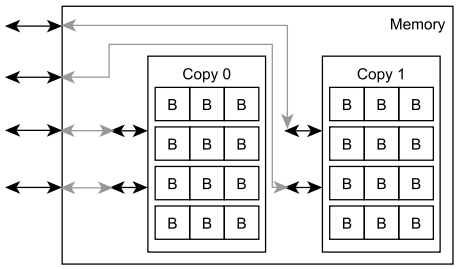

Fig. 1. The memory architecture of simple replication approach.

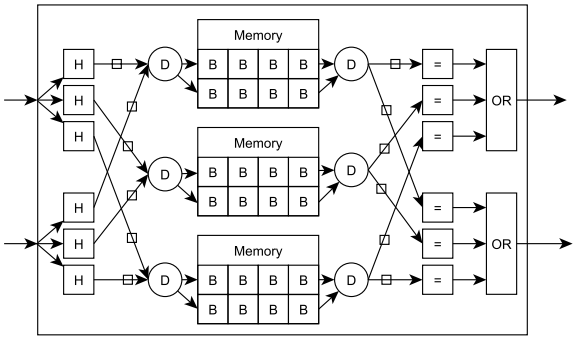

Fig. 2. The top-level architecture of the proposed optimization.

point, after which the frequency cannot be further increased due to the limitations of used FPGA technology. The other way to increased throughput is through a design of the new architecture of cuckoo hashing that can carry out more than one rule lookup per each clock cycle. This translates to more than one memory access per clock cycle to each utilized hash table. Current Xilinx FPGAs use BlockRAM tiles as the main type of on-chip memories. These have 2 independent ports, therefore we can easily perform 2 memory accesses per clock cycle with no additional cost.

In order to enable more than 2 accesses per clock cycle, we can simply replicate the memories. This is illustrated for 4 accesses in the figure 1. Two of the accesses are mapped to one copy of the memory and two are mapped to the other copy. This approach is not particularly efficient because we need to double the memory in order to achieve doubled throughput. However, we can leverage the internal structure of FPGAs and their memory tiles. A single copy of a larger memory is internally usually composed of more than one BlockRAM tile (B blocks). Each BlockRAM on current Xilinx chips [23] can be used as 36 b wide dual port memory with 1 024 entries. Larger memories are constructed utilizing multiple BlockRAMs organized into several rows and columns. For example, figure 1 corresponds to data width of up to 108 b and 4 048 entries.

A. Proposed Approach

If we already have more than one row of BlockRAMs in each table we should be able to do more than just two memory accesses per clock cycle. We can, in fact, ideally do two accesses per cycle independently to each of the individual rows. This fact can be leveraged to optimize the previously mentioned simple replication approach. We propose an FPGA architecture of cuckoo hashing shown in figure 2. The proposed approach is also applicable to any other kinds of hash tables, but we choose cuckoo hashing as it is the most effective existing hashing scheme.

The figure shows the architecture able to carry out up to 2 parallel lookups per cycle with cuckoo hashing using 3 different hash functions/tables. The memory blocks used here are similar to the blocks from figure 1 – meaning that they internally consist of multiple independent rows of BlockRAMs. Hash functions are computed for each lookup key (H blocks) and are connected to a distribution logic (D blocks). There is one distributor block for each hash function/table of the cuckoo hashing. The distributor consists primarily of logic that maps the requested memory accesses into corresponding table rows given by a few most significant bits of their hash values and distributes them onto available BlockRAM ports for each of these rows. On the other side, it correctly forwards read data from each memory row and port to the corresponding comparison logic (= and OR blocks). The basic idea is to replicate memory fewer times than in the case of a simple approach (fewer replicas than required parallel packet lookups) as we can perform multiple accesses per clock cycle simply by hash functions that are pointing into different table rows. Additionally, memory can also be replicated here to enable more than two parallel access ports for each row.

So, the distributor blocks determine which row of the BlockRAMs is accessed by each lookup and sets the control logic in a way to carry out all the lookups that are not in conflict with each other. Conflicts, in this case, mean that there are more lookups wanting to access the same row of one table than there are available access ports in this row. Note that since the memories can still be replicated the number of available access ports might be higher than two. All the lookups that could not be carried out in the first cycle will be carried out in consecutive cycles until all of the requested lookups are finished. This means that the lookup of all of the inputs might take more than one cycle. However, the basic idea is that the relative number of occurring conflicts (or rather number of additional cycles needed) is pretty low for higher numbers of memory rows, thus reducing throughput only slightly. Compared to that, the saved memory resources thanks to no or weaker memory replication are considerable.

For example, consider a case where four lookups each clock cycle are needed and there are four rows of BlockRAMs with only two ports each (meaning no memory replication). There are no conflicts unless at least three of the four lookups need to access the same row. In case of the conflict, two of the conflicting accesses can still be carried out together with all others that are not in conflict. The last one or two

accesses from the conflicting group has to be carried out in the next cycle. Even if there is a conflict every time, we still achieve the same throughput as the simple architecture with the same memory requirements (replication factor). In the example without replication, we would do four lookups in two clock cycles which is the same as the simple approach with two lookups each cycle. This shows that at worst the proposed solution is on par with the simple solution in terms of both memory and throughput. However, the key idea is that the conflicts do not occur each time and are actually pretty infrequent (20 % conflict chance in this example), therefore the effective achieved throughput is considerably better.

It is also important to note that we can easily achieve independence in the conflict handling for each parallel hash table used in the cuckoo hashing. The distributor corresponding to a single hash table does not need to wait until all the other distributors carried out all their lookups. Instead, there are small input and output buffers that are used to synchronize the results (denoted by squares on corresponding connections in figure 2). This makes the architecture a lot more efficient as the throughput is not governed by the probability of no conflicts in all of the tables together but rather by the probability that there are no conflicts in every single table independently. This independent probability is a lot lower especially when a higher number of parallel hash tables are used.

Of course, the described buffers consume some additional FPGA resources. Also, the distributors themselves introduce some logic overhead compared to simple replication approach. In the simple approach, there is a dedicated port for each parallel lookup, therefore hash functions (inputs) and comparison logic (outputs) can be directly connected to appropriate memories without the distributors. The core of each distributor is a planner, that can evaluate and resolve access conflicts – basically a group of encoders and decoders to select a valid access plan for each cycle. The planner controls two columns of multiplexers: the first to route planed access requests to correct memory rows/ports and the second to pair read data with their corresponding requests. Additional registers are used to thoroughly pipeline the distributors for better frequency and to correctly synchronize all operations together. The total FPGA logic overhead of the distributors and buffers is expected to be manageable compared to complex hashing blocks which are usually considerably large and contain critical paths.

The described architecture can be optimized for better throughput even further as during conflicts the available access ports of memories are currently not fully utilized in the added clock cycles. For example, if only one lookup cannot be carried out in the first cycle it has to be carried out in the second (additional) one. Reserving one full clock cycle just for one extra lookup is inefficient. A more reasonable approach would be to combine the extra lookup cycles with some of the lookups needed for the next inputs. While this cycle sharing increases the throughput by a small percentage it requires a lot more complex distributor and buffer architectures. Because of this the rest of the paper deals with the architecture without such optimization.

B. Analysis of Conflicts

It is possible to mathematically analyze the probability of conflict occurrence and derive the achievable throughput of the proposed architecture with given parameters. There are 3 main parameters of the architecture when it comes to the probability of conflicts: the number of rows of BlockRAMs in each table r, the number of parallel lookups per clock cycle l corresponding to the number of inputs, and the number of available access ports for each table row p.

The probability that a single lookup needs to access one specific selected row and the probability that it needs to access any other row are complementary:

$$p_s(r) = \frac{1}{r} \tag{1}$$

$$p_{ns}(r) = \frac{r-1}{r} \tag{2}$$

First of all, for any given n the probability that exactly n lookups out of total l in one cycle need to access one selected row out of r rows can be computed as a product of: the probability that selected n lookups access selected row, the probability that all the other $l-n$ lookups do not access this row, and the number of combinations by which it is possible to position those n lookups into all l. The appropriate equation:

$$\begin{aligned} p_s(n,l,r) &= (p_s(r))^n * (p_{ns}(r))^{l-n} * \binom{l}{n} \\ &= \left(\frac{1}{r}\right)^n * \left(\frac{r-1}{r}\right)^{l-n} * \binom{l}{n} \end{aligned} \tag{3}$$

To get the probability that any of the rows will have exactly n lookups mapped onto it we simply multiply the previous probability from equation 3 by the number of rows:

$$\begin{aligned} p_a(n,l,r) &= p_s(n,l,r) * r \\ &= \left(\frac{1}{r}\right)^{n-1} * \left(\frac{r-1}{r}\right)^{l-n} * \binom{l}{n} \end{aligned} \tag{4}$$

Now to approximate the probability that more than n lookups out of all l in one cycle need to access the same row out of r we can simply sum the probabilities from equation 4 for all values higher than given n:

$$\begin{aligned} p_{c,a}(n,l,r) &= \sum_{i=n+1}^{l} p_a(i,l,r) \\ &= \sum_{i=n+1}^{l} \left(\frac{1}{r}\right)^{i-1} * \left(\frac{r-1}{r}\right)^{l-i} * \binom{l}{i} \end{aligned} \tag{5}$$

This sum does not account for the fact that solution spaces described by some of the summed probabilities have non-empty intersections with one another (some conflict variants are counted multiple times). To counter this fact we would have to compute probabilities that exactly n lookups will be mapped onto the same row while there is no other row with n or more lookups mapped onto it. This leads to exponentially more complex nested sums. However, the approximate results achieved by the equation 5 are always higher than the actual

results, which means they would actually give us more pessimistic results for the throughput. Additionally this approximation is very precise for results under configurations that are the most interesting for us. For example, it is absolutely precise if p is higher or equal to $l/2$, since in this case, it is impossible for two different rows to have more than p accesses mapped at the same time.

The equation 5 essentially approximates the probability that there will be a conflict for architecture with *l* lookups, *r* rows of BlockRAMs and *p=n* ports for each row. However, not all conflicts are equal when it comes to their effect on the achieved throughput. For example, if $p = 2$ and 6 lookups need to access the same row it takes 3 cycles to carry out all of all them, while when 4 lookups need to access the same row only 2 cycles are needed. To extend our equations and reflect this we use a weighted sum:

$$c_{w,c}(n,l,r) = \sum_{i=n+1}^{l} w(i,n) * p_a(i,l,r) \tag{6}$$

The weight w here represents the number of cycles needed to resolve the conflict in each case:

$$w(i,n) = \left\lceil \frac{i}{n} \right\rceil \tag{7}$$

Finally we can do one last thing to get how many times more cycles (on average) are needed compared to the case without any conflicts. The equation 6 sums only weighted probabilities of conflicts. We need to add also the probability that there will be no conflict. Weight corresponding to no conflict is obviously 1 since even when there is no conflict we still need one clock cycle to carry out all the lookups. So the coefficient that gives us the relation between needed cycles (achieved throughputs) is computed as follows:

$$c(n,l,r) = c_{w,c}(n,l,r) + (1 - p_{c,a}(n,l,r)) \tag{8}$$

In conclusion, the proposed optimized architecture with l lookups, r BlockRAM rows, and p ports can achieve throughput equivalent to an average of m lookups per cycle, where:

$$m = \frac{l}{c(p,l,r)} \tag{9}$$

Thanks to the previously mentioned buffers there is no need to include number of hash functions (parallel hash tables) into our computations. Logic and memories corresponding to each hash operate independently of one another and their results are only synchronized afterward via buffers. This means that if there is a collision in memory tied to one hash another hash with no collision does not have to wait.

IV. Results

The results in this section are obtained through the previously mentioned mathematical analysis and are confirmed through experiments with implemented architecture. Measurements are based on design synthesis for the Xilinx UltraScale+ XCVU9P FPGA [23] using Vivado 2017.4 tool. The architecture is able to achieve working frequency (F_{max}) of more than 400 MHz for every evaluated configuration. Therefore, the following throughput results are all shown for 400 MHz clock frequency. All the cases used 104 b wide key that is sufficient for the classification of standard IPv4 flows (5-tuple) and 32 b wide arbitrary data (action). There are 3 main parameters that are worth exploring in the results – resource requirements (BRAMs, LUTs), achievable throughput (lookups per cycle, Mpps, Gbps), and effective rule capacity.

Hash functions	Rows of BRAMs	Total capacity	Effective capacity
3	1	3 072	2 765
3	2	6 144	5 530
3	4	12 288	11 059
3	8	24 576	22 118
3	16	49 152	44 236
4	1	4 096	3 891
4	2	8 192	7 782
4	4	16 384	15 565
4	8	32 768	31 130
4	16	65 536	62 260

TABLE I
CAPACITIES OF CUCKOO HASHING FOR DIFFERENT PARAMETERS.

Table I shows different capacities of cuckoo hashing architecture based on the number of hash functions and the number of BlockRAM rows for each table (each row has 1 024 items). Total (theoretical) capacity and achievable effective (mean) capacity are shown. For three functions the efficiency of capacity utilization is around 90 %, for four it is around 95 %. This is consistent with similar measurements in [21]. Table I is primarily used to illustrate cuckoo hashing capacities that are considered in the evaluation.

Figure 3 captures the relation between throughput and memory requirements of architectures with three hash functions in different configurations. Lines in the graph represent throughput and memory requirements of simple memory replication approach for a different number of BlockRAM rows used. Again, the number of rows is directly tied to the capacity of the architecture as shown in table I. These results form a baseline for evaluation of the designed optimization.

Each point in the graph shows results for a different configuration of the proposed memory optimized approach. The color of a point represents the number of BlockRAM rows used (the capacity of the architecture) and its shape represents how many lookups (number of inputs l) the architecture supports. Our approach is clearly better in terms of used memory for each given throughput achieved as all points are below lines of appropriate color. Obviously, when there is only one row of BlockRAMs (black line) there is no possibility to employ our optimization and gain something. However, even when there are only 2 rows of BlockRAMs (light blue) we can already achieve better results. For example, using an architecture with 10 lookups (full circles) we can achieve 48.5 % increase in throughput without any memory duplication.

The results tend to get even better when using more rows of BlockRAMs. This is expected behavior since more rows mean

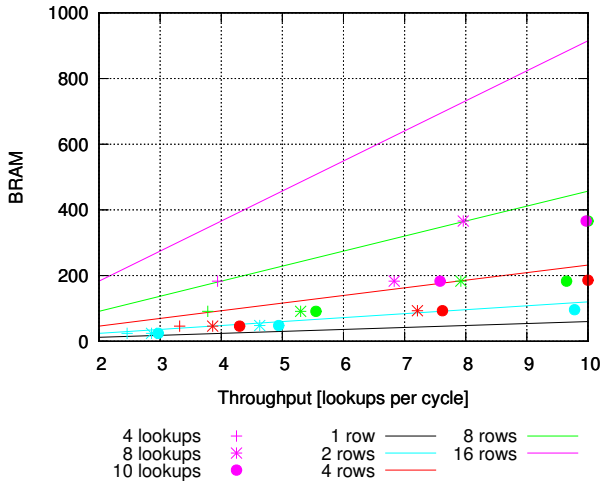

Fig. 3. The relation between memory and throughput for 3 hash functions.

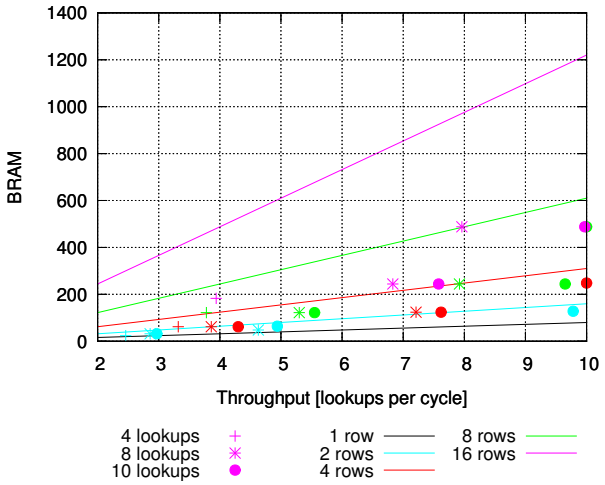

Fig. 4. The relation between memory and throughput for 4 hash functions.

more chance for the lookups to be better spread out between different rows, thus the probability of conflicts decreases. In case of 16 BlockRAM rows (pink), it is possible to achieve nearly twice the throughput without any memory duplication even when using the architecture with only 4 lookups (cross). If we use architectures with 8 (star) or 10 (full circle) lookups the speedup is even further amplified and nearly 7 or 7.5 times higher throughput can be achieved with no additional memory requirements. Additionally, with two times replicated memory, we can achieve nearly the full throughput of 10 lookups per cycle. This means that we can achieve 99.7 % of throughput with only 40 % of used memory.

The number of hash functions has no effect on the efficiency of the proposed optimization approach (only on the efficiency of cuckoo hashing itself). This can be clearly seen by comparing figure 3 with figure 4. Figure 4 shows the relation of utilized memory and achieved throughput for different architecture configurations with 4 hash functions. Graphs shown by figures 3 and 4 are pretty much the same only shifted slightly along the y-axis. The increase in memory requirements is offset by the higher capacity of the architectures (see table I).

Figures 3 and 4 might suggest that architectures with more lookups (inputs) are always better. However, this is not the case when it comes to utilized on-chip logic resources. Architecture with more lookups needs more hash function computations, more buffers, and larger distributors. The relation between on-chip logic, more specifically required LUTs, and throughput for 3 hash functions is illustrated by figure 5. The graph shows that if we use an architecture with for example 10 lookups (pink) the logic requirements go up together with the level of memory duplication and the achieved throughput. Memory-optimized architecture with 10 lookups, 16 rows and 4 memory ports (two memory replicas) achieves 99.7 % of throughput requiring only 40 % of memory at a cost of 466 % of LUTs compared to the simple approach with 10 lookups

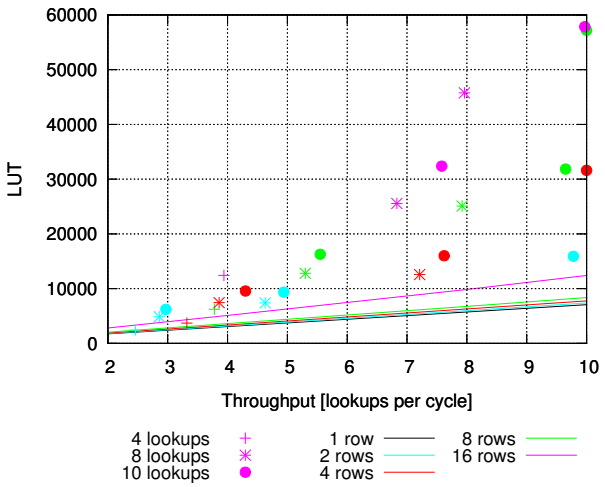

Fig. 5. The relation between used logic and throughput for 3 hash functions.

and 16 rows. From a different point of view, the optimized architecture with 10 lookups, 2 rows and 2 memory ports (no replication) achieves 48.5 % increased throughput requiring the same memory at a cost of 244 % increase in LUTs compared to the simple approach with 2 lookups and 2 rows. However, we argue that the decreased memory requirements or increased throughput, depending on the way we look at it, is a favorable trade-off for the increase in on-chip logic. In many cases, even the increased logic requirements are still feasible for current FPGAs (only a few percents of the total available), while increasing the throughput without the need to replicate memories can prove to be more critical.

In order to better illustrate the impact of the proposed memory optimization on the achieved results, we analyze them from different views. First of all, let's take a look at the best results that we can achieve if we want to reach a given minimal

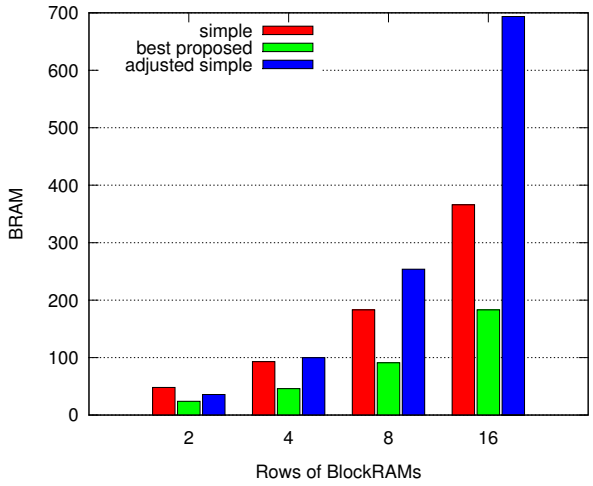

Fig. 6. Memory requirements comparison when achieving at least 800 Gbps.

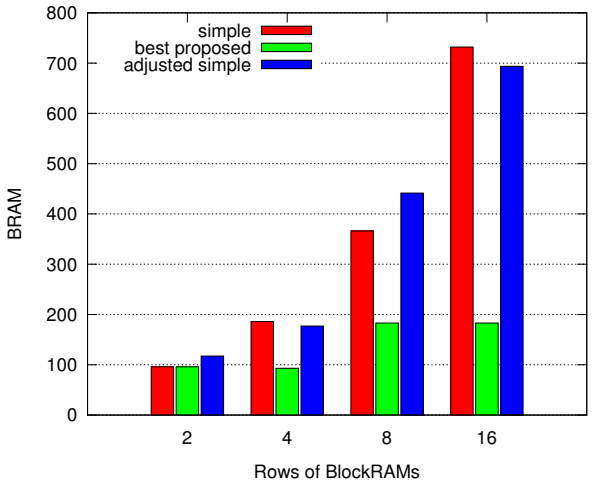

Fig. 7. Memory requirements comparison when achieving at least 1.6 Tbps.

throughput. Figures 6, 7, 8 illustrate the memory requirements of the best configurations of the proposed approach (green) compared to the baseline given by the simple memory replication (red) when we want a throughput of 800 Gbps, 1.6 Tbps or 2.4 Tbps. The best configuration is the one that requires the least memory while still satisfying the minimal throughput threshold. This obviously means that actually achievable throughputs of compared simple and optimized configurations are not the same. For better comparison, we can leverage the fact that memory of the simple approach scales linearly with throughput and adjust the required memory to the point where the simple approach has exactly the same throughput as the optimized (blue). We can see that our approach is more and more effective as the total capacity of the cuckoo hash table rises. For 2 rows it is possible to achieve the same throughput as simple replication with somewhere between 67 % and 80 % of required memory (after adjustment), while for 16 rows only between 25 % and 40 % of memory is needed. To be more precise the most significant factor that governs how much memory can be saved is the ratio between the number of rows (capacity) and required throughput (parallel lookups). The higher the capacity the better the results become as the lookups can be spread among more rows.

On the other hand, we can analyze the achievable throughput for a given number of BlockRAMs (e.g. 200) that we have available. The best cases of the proposed optimized approach are obtained when using architectures with 32 lookups. This is chosen mainly because for 2 rows of BlockRAMs the memories can be duplicated up to 8 times in the simple solution, which means 16 lookups. Therefore to obtain reasonable results we chose architectures with at least twice as much lookups. The results are shown in figure 9. An interesting thing can be observed: even as the number of rows (and therefore capacity) increases and the duplication factor decreases the throughput of the proposed approach stays relatively the same.

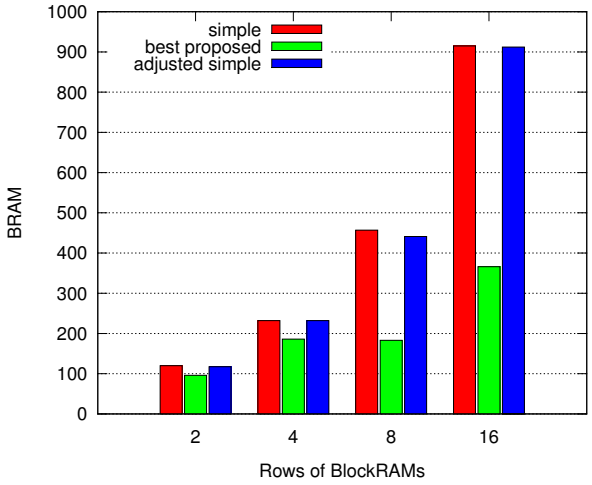

Fig. 8. Memory requirements comparison when achieving at least 2.4 Tbps.

This is caused by the fact that while the number of copies of memories (memory access ports) decreases it is balanced by a better spreading of all the lookups between more rows of independently operating BlockRAMs. If we choose other numbers of BlockRAMs the observed trend is pretty similar.

V. Conclusion

The paper presents novel memory efficient hardware architecture for exact match packet classification at very high speeds (400 Gbps and beyond) using the cuckoo hashing algorithm. The proposed architecture offers an easily configurable tradeoff between achieved throughput, required memory, utilized logic, and rule capacity. With the proposed optimization, it is possible to implement exact match packet classification for large rulesets operating at very high throughputs with efficient utilization of available memory. There are several

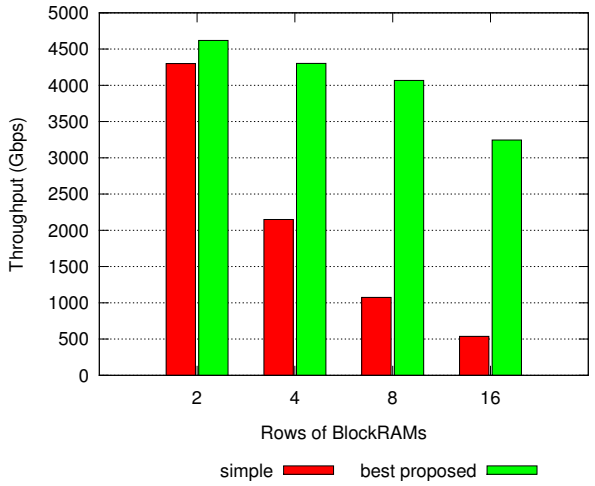

Fig. 9. Maximal throughputs of evaluated approaches for 200 BlockRAMs.

ways in which the architecture can be used – either to maximize throughput and rule capacity on devices with limited memory resources or to minimize memory requirements while satisfying needed rule capacity and throughput.

Experimental results with the proposed simple and optimized architectures of cuckoo hashing show few interesting facts. First of all the optimized architecture is considerably more memory efficient than a simple replication approach. With correct configuration, we are able to achieve 99.7 % of throughput for only 40 % of required memory compared to the simple approach. If the required rule capacity of the architecture is high enough our optimized approach is generally able to retain the same throughputs with only 25-40% of memory utilized compared to the simple solution. This way we can achieve an unprecedented throughput of 2.4 Tbps and effective capacity of over 44 000 IPv4 5-tuple (flow) rules for the cost of only 366 BRAMs. The only downside of the proposed optimized architecture is increased requirement of on-chip logic. However, we argue that the benefits of decreased memory requirements and increased throughput outweigh this issue in most practical cases.

Acknowledgments

This research is supported by the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 by the MEYS of the Czech Republic; the IT4Innovations excellence in science project IT4I XS – LQ1602; and by the Ministry of the Interior of the Czech Republic projects VI20172020064 and VI20152019001.

References

[1] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[3] P. Benáček, V. Puš, and H. Kubátová, "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.

[4] P. Benáček, V. Puš, H. Kubátová, and T. Čejka, "P4-to-VHDL: Automatic generation of high-speed input and output network blocks," *Microprocessors and Microsystems*, vol. 56, pp. 22 – 33, 2018.

[5] D. Taylor and J. Turner, "Scalable packet classification using distributed crossproducing of field labels," in *24th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2005, pp. 269–280.

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

[7] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2003, pp. 213–224.

[8] H. Lee, W. Jiang, and V. K. Prasanna, "Scalable High-Throughput SRAM-Based Architecture for IP Lookup Using FPGA," in *International Conference on Field Programmable Logic and Applications*, 2008.

[9] 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*. New York, NY, USA: ACM, 2006, pp. 61–70.

[10] 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.

[11] J. Kořenek, V. Puš, and J. Blaho, "Memory optimization for packet classification algorithms," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ser. Association for Computing Machinery. Association for Computing Machinery, 2009, pp. 165–166.

[12] 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, Jul. 2012, ISSN 0018-9340.

[13] Y. Qi, J. Fong, W. Jiang, B. Xu, J. Li, and V. Prasanna, "Multi-dimensional packet classification on fpga: 100 gbps and beyond," in *2010 International Conference on Field-Programmable Technology*.

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

[15] 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, 2005, pp. 238–245.

[16] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 191–202, 1998.

[17] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, 2012.

[18] M. Kekely and J. Korenek, "Packet classification with limited memory resources," in *2017 Euromicro Conference on Digital System Design*. Institute of Electrical and Electronics Engineers, 2017, pp. 179–183.

[19] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms - ESA 2001*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, vol. 2161, pp. 121–133.

[20] A. Kirsch, M. Mitzenmacher, Y. Baohua, X. Yibo, and L. Jun, "Using a queue to de-amortize cuckoo hashing in hardware," 2007. [Online]. Available: http://www.eecs.harvard.edu/~michaelm/postscripts/aller2007.pdf

[21] L. Kekely, M. Žádník, J. Matoušek, and J. Kořenek, "Fast lookup for dynamic packet filtering in FPGA," in *17th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. Warszaw, Poland: IEEE Computer Society, 2014, pp. 219–222, iSBN: 978-1-4799-4558-0.

[22] L. Kekely, J. Kucera, V. Pus, J. Korenek, and A. V. Vasilakos, "Software defined monitoring of application protocols," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 615–626, Feb. 2016.

[23] Xilinx, *UltraScale and UltraScale+ FPGAs Packaging and Pinouts*, Xilinx Inc., 2016, UG575.

A.4 Paper 4

Pipelined ALU for Effective External Memory Access in FPGA

2020 23rd Euromicro Conference on Digital System Design (DSD)

Pipelined ALU for effective external memory access in FPGA

Tomáš Beneš
Faculty of Information Technology
Czech Technical University
Prague, Czech Republic
benesto3@fit.cvut.cz

Michal Kekely
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
ikekelym@fit.vutbr.cz

Karel Hynek
Faculty of Information Technology
Czech Technical University
Prague, Czech Republic
hynekkar@fit.cvut.cz

Tomáš Čejka
CESNET, a.l.e.
Prague, Czech Republic
cejkat@cesnet.cz

***Abstract*—The external memories in digital design are closely related to high response time. The most common approach to mitigate latency is adding a caching mechanism into the memory subsystem. This solution might be sufficient in CPU architecture, where we can reschedule operations when a cache miss occurs. However, the FPGA architectures are usually accelerators with simple functionality, where it is not possible to postpone work. The cache miss often leads to whole pipeline stall or even to data loss. The architecture we present in this paper reduces this problem by aggregating arithmetic operations into the memory subsystem itself. Fast data processing is achieved because arithmetic operations working with external data are offloaded. Our architecture reaches a speed of 200 Mp/s (operations carried out). It is designed to be used in systems with link speeds of 100 Gb/s. It outperforms other implementations by a factor of at least 3. The additional benefit of our architecture is reducing the number of memory transactions by a factor of two on real-world datasets.**

***Index Terms*—cache, external memory, FPGA, network monitoring**

I. Introduction

With the rise of high-speed networks, video processing, and high-speed storage systems, the FPGA systems are becoming more widespread due to their variability and their computational performance. Specialized implementations inside FPGA outclasses the general-purpose CPUs in orders of magnitude. These systems are processing a massive amount of data, which usually needs to be temporally stored for the processing to take place. FPGAs have internal memories such as BRAMS, URAM, or distributed memories. However, these have sizes in the order of MB, which is not enough when tables tracking flows are used. Therefore, the designers have to use external memory like static random access memory (SRAM) or dynamic random access memory (DRAM).

The massive amount of storage space provided by the external memories comes with the cost of high access time and usually non-constant access time [1]. This disadvantage is the main reason why most of the designs using these memories tend to be very complicated and poorly maintainable.

The main principle which is used to handle access time is called pipelining. It is a well-known, efficient, and effective way of increasing the throughput and overlapping the different access times. The pipelining is usually sufficient for accessing data inside the external memory. However, it does not apply when Read-Modify-Write operations take place. These operations introduce various hazards such as read after write (RAW), write after read (WAR) and write after write (WAW). These hazards always need to be addressed when designing a processing pipeline accessing shared memory resources. While the design is trivial for low latency memories it becomes more complicated with introduction of external memories.

The P4 language is an elegant way of describing packet processing, which can be implemented on multiple platforms, e.g., CPU [2], FPGA [3], [4] or ASIC [5]. Having a P4 compiler for FPGA provides a combination of flexibility and speed. The P4 program can be easily uploaded to the FPGA chip, which is capable of processing high-speed backbone network traffic with up to 100 Gb/s speed.

One of the problems is that many application which can be implemented in P4, such as Network monitoring, or Virtual switching is the previously mentioned lack of high capacity memories on todays FPGA chips [6], [7]. This requirement for the FPGA architectures leads to use of external memories. The memory needs to be used for classification rules and also for storing stateful and statistical information (byte and packet counters) which introduces Read Modify Write operations.

Basic caching mechanisms are in this case insufficient since they are based on the locality principle and work best only when the subsequent memory accesses are using the same small regions of the memory. Žádník et al. [8] showed that network flows are usually distributed pretty randomly, and the locality principle falls apart. Therefore, we need a unit that optimizes memory accesses in such a way that the Read Modify Write operations ideally do not require pipeline stalls even when memory accesses are randomly distributed over the whole external memory.

In this paper, we propose an architecture for the implementation of pipelined designs using external memories. The main focus of this architecture is to simplify the design process, reduce utilization of the external memories by aggregating

978-1-7281-9535-3/20/$31.00 ©2020 IEEE
DOI 10.1109/DSD51259.2020.00026

operations per memory transaction, and solve memory hazards present in standard designs.

II. State of the art

The basic idea of caching has been well known for decades and widely researched for use with processors. Different hierarchies of caches are used in today's processors, but as we previously mentioned these cache hierarchies relay on the locality of reference principle. The lower the locality of memory accesses the bigger cache is needed. One of the approaches trying to solve this problem is to keep only a small number of distinct tags of cached data (Caching Address Tags or CAT cache [9]). This lowers the resources needed by the cache, while not hurting the performance in any significant way. However, the locality principle still needs to be satisfied in order for this approach to be efficient.

A whole other area of caching is used when it comes to the World Wide Web [10]. There are some interesting ideas that can be leveraged such as keeping the entries that were accessed the most in the most recent time window (and therefore are most likely to be accessed again). This helps alleviate some of the problems with the reference locality.

New development of in-memory computation or in-memory processing [11] [12] [13] aims to deviate from the standard architectures where the memory (storage) and processing are separate. Separate memory and processing require a lot of data to be transferred back and forth between processing and storage units. This new approach tries to decrease the amount of this data by moving some (or all) of the functionality directly to the memory or storage units. These ideas can be applied to FPGA and external memories by pushing a lot of the functionality that modifies the memory entries as close to the memory as possible.

All of the current approaches either still rely on the locality of the data to be efficient or are not feasible for an FPGA implementation. A new approach is therefore needed to address all of the issues outlined previously.

III. Architecture

The main principle of the proposed architecture is to separate the design into two parts The first part is the control logic which represents the series of operations that need to be executed on the data inside the external memory. The second part is the operational logic (ALU), which handles the operations or commands on the external memory (such as Read, Write, Add, Sub, XOR). The ALU tries to aggregate the operations by their associativity, which reduces the number of memory accesses needed compared to a standard implementation. The proposed architecture also minimizes the access time of the external memory by placing the ALU as close as possible to the external memory.

The architecture of the proposed solution is shown in Figure 1. It has two main interfaces. Both of the interfaces are a basic request/response interfaces, one towards the external memory (this interface request reads/writes on the external memory) and one towards the Control Logic (reads or different operations are requested via this interface). The architecture

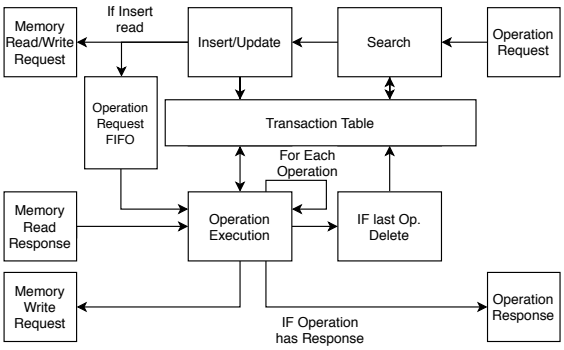

Fig. 1. Diagram of the ALU functionality

expects the requests to the memory to be carried out in order. The basic idea is that whenever request from the Control Logic arrives an appropriate entry in the Transaction Table is looked-up. If such entry does not exist, it is created. Every operation and request that matches this entry is then carried out on top of this entry. If all of the pending operations on entry have been executed, the entry can be deleted (freed), and a different entry can take its place.

A. Principle of the operation aggregation

The core of our proposed architecture is the Transaction Table. It is very similar to tables found in traditional caches. However, Transaction Table in our approach can contain multiple entries with the same address. Each entry in the Transaction Table consists of multiple fields. This is illustrated by Table I. The *Flags* field signalizes if the entry is empty and if it is deprecated, *Address* holds the index to the memory that this entry represents (this value is used for look-ups), *Operation count* indicates the number of reads requested from this memory index and is also an index into the array of accumulators, *Array of accumulators* holds multiple accumulator values, each accumulator holds and aggregated value of possibly multiple operations. Finally, the *Memory value* field holds the value read from the memory (once its read).

TABLE I
Description of Transaction table Entry

Flags	Addr.	Op. Count	Acc_0	...	Acc_n	Memory Value

The aggregation uses the associativity property of the operations. For example, lets assume this example of operations carried out on the data inside external memory on the same memory address: o_1, o_2, r_1

r_x represents read operation, and o_x represents operand in the operation $\odot$ which should be performed. After the application of these operations the outputs of the reads are represented by r_1 where m represents the value in the memory before any of these operations were applied. Leveraging the associativity property and the ability to store values in accumulator fields we can rearrange the values as follows:

$$r_1 = m \odot o_1 \odot o_2 \qquad r_1 = m \odot (o_1 \odot o_2)$$
$$acc_0 = (o_1 \odot o_2) \qquad r_1 = m \odot acc_0$$

Using this approach allows us to aggregate multiple operations requested on the same memory location into a single

memory transaction which allows us to execute multiple operations before the actual value from the memory is read. This separates the actual memory accesses from the operations. The operations can then be carried out in the entries of the Transaction table (partial updates inside acc_x).

B. Behaviour

Different operations (or requests or responses that arrive) need to be handled, for simplicity let's call these events. Each event has to be handled in a way that keeps the state of the memory consistent. The main events that can occur are request from the Control Logic to carry out operation $\odot$ on address A with operand V ($\boldsymbol{\odot(A,V)}$), request from the Control Logic to read data from address A ($\boldsymbol{read(A)}$), request from the Control Logic to write data V to address A ($\boldsymbol{write(A,V)}$) and response from the memory with the data V from address A ($\boldsymbol{mem\_resp(A,V)}$).

An abstract behavior of responding to event $\boldsymbol{\odot(A,V)}$ is to find a corresponding entry based on address A (if one exits) or create a new one (and possibly wait and stall the pipeline if no free entries are available). When an entry is created it is initialized and a read from the memory is also issued. Once we have an entry we simply update the last accumulator value (pointed to by operation count) with the $\odot V$.

In a similar fashion a reaction to event $\boldsymbol{read(A)}$ is to find the appropriate entry (or create one), possibly wait if the found entry has no accumulators left (operation count cannot be incremented) and then just increment the operation count and set the next accumulator (pointed to by the incremented value of operation count) the the value of the previous one ($acc_{new\_op\_count} = acc_{new\_op\_count-1}$).

When event $\boldsymbol{write(A,V)}$ occurs we simply find a corresponding entry and deprecate it. Deprecated entry is not being matched anymore because the write reset the value associated with this memory index. However, we do not free the entry yet since it can have some of the reads still pending (these reads were issued before the write and therefore they should still use the old value from the memory). Finally a write request to the memory is issued (with value V to address A).

When an memory response arrives (event $\boldsymbol{mem\_resp(A,V)}$) we choose the entry tied to this memory read (for example via FIFO of identifiers of memory reads). Within this entry the memory value field is set to V and for each valid accumulator (given by the operation count) send a response to Control Logic - $r_i = m \odot acc_i$. Also if the entry was not yet deprecated we issue a write into the memory updating it with the last accumulator value - $m_{new} = m \odot acc_{op\_count}$. After this the entry is freed.

C. Parameter analysis

The main parameters of the architecture are the number of entries in the Transaction Table and the number of accumulators within each entry. Setting the parameters too low leads to decreased throughput (more likely stalls due to no available entry), setting them too high leads to increased resource utilization.

TABLE II
RESOURCE UTILIZATION AND FREQUENCY OF THE PROPOSED ARCHITECTURE.

Size	LUTs	Registers	Frequency
4	2849	1439	+350 MHz
8	4151	2484	+350 MHz
16	7862	4573	285 MHz
32	16241	8750	225 MHz
64	31491	17103	188 MHz

The optimal number of entries and accumulators should be set based on multiple criteria - latency of the memory, required throughput, properties of the memory operations. However, the latency of the memory gives us how many requests can arrive before any of the requests was actually carried out and therefore the required number of entries and accumulators. Either all of the events used different addresses which means we need separate entry for each, or they used the same address and we need one entry with possibly one accumulator for each.

IV. RESULTS

The architecture described in the previous sections was implemented and the results presented in this section were obtained. Measurements of FPGA resources requirements for Xilinx are based on design implementations for the UltraScale+ XCVU7P chip [6] using Vivado 2019.1 tool.

Basic resource utilization of the architecture for the different number of entries and accumulators is shown in Table II. The configuration that was used worked with the 64b wide memory words. As expected the resource utilization goes up pretty linearly and is still manageable even for tens of entries. However, the maximal frequency slowly decreases. Our results in Table III shows aggregation 16 or 32 entries should be sufficient, which still satisfies at least the frequency of 200 MHz which means possibly up to 200 Mp/s (or in this case 200 million operations carried out). This is more than enough for packet processing at the line rate of at least 100 Gb/s.

Real network traces used for this evaluation were obtained from the high-speed backbone network managed by CESNET. CESNET is the Czech National Research and Educational Network operator with an infrastructure consisting of multiple optical links with bandwidth up to 100 Gb/s. This optical network serves around 200 000 users and routes mainly IP traffic. Data traces were captured at different points of the network. The captured traces contain both IPv4 and IPv6 flows, with IPv4 dominating. In the following evaluations, we used two traces: *meter1* and *meter4*. Both traces contain 1,000,000 packets captured during different periods of day.

We evaluated the architecture under multiple cases, where the memory addresses accessed were based on different identifiers from the network data. There are 2 main parameters that are worth exploring in the obtained results — number of memory accesses and number of operations carried out.

We compared our approach to three other possible implementations designed to fulfill the same functionality. First of all a Trivial implementation using sequential automaton. It issues a memory transaction for each requested operation and

waits for the result to update it a write it back. The second approach is Trivial Cache using the same sequential automaton as the Trivial implementation extended with traditional memory cache. The final approach is Trivial Pipelined design with sufficient stages to cover the memory latency. However, it stalls when a collision occurs (2 accesses to the same memory address) to prevent data hazards.

The comparison of the architectures consists of two results. First is the time (clock cycles) needed for processing the whole input dataset. This is shown as a ratio between a theoretical limit of a number of clock cycles for processing (based on number of samples in the dataset) and the actual number of the clock cycles it took each implementation. The second is the number of memory transactions needed for processing the whole input dataset. This is represented in the same way as the first result.

The Table III shows both results of the compared implementations. It is shown that the Trivial solution is not affected by the diversity of the datasets as expected. It also shows the upper bound for the clock cycles needed to process the whole dataset.

The Trivial Cache implementation using traditional memory cache is showing a very promising result on non-randomized datasets. It also shows a significant reduction of memory transactions with non-randomized datasets.

The Trivial Pipeline implementation has minimal stalling issues when it comes to randomized data, however, its performance suffers, when it comes to non-randomized datasets.

Our architecture out-performers all of the compared implementations and reaches almost our theoretical limits for almost every dataset, which translates to full throughput for a given frequency. Only the results for the random dataset are close to equal with the Trivial Pipeline implementation because our architecture uses a limited transaction table. When the transaction table is saturated our design begins to stall and is downgraded to the trivial pipelined implementation. For datasets from real-world application our architecture outperforms every other implementation at least by a factor of 3 when it comes to throughput. Another benefit of our architecture is the memory transactions reduction which is on our real-world datasets by a factor of 2.

TABLE III
COMPARISON OF RESULT FOR MODELS

Test	Trivial	T. Cache	T. Pipeline	Our arch.
rnd_count	0.125-1	0.138-0.936	0.501-1	0.500-1.000
dst_IP	0.125-1	0.391-0.546	0.221-1	1.000-0.432
src_dst_IP	0.125-1	0.358-0.565	0.221-1	1.000-0.433
full_flow	0.125-1	0.342-0.576	0.223-1	1.000-0.438

V. CONCLUSION

This paper presents and examines the design of a novel caching FPGA architecture for aggregating associative operations. The proposed architecture offers a trade-off between increased resource utilization and increased number and speed of operations that are carried out. We are able to update monitoring statistics (counting of the number of packets or bytes in a flow) for network traffic at the line rate of 100 Gb/s even for a big number (millions) of different flows thanks to optimal usage of external memories that the proposed architecture enables. Compared to trivial approaches, we are able to increase the throughput by a factor of 3 and reduces the number of memory transactions by a factor of 2 at a cost of between 2,849 and 16,241 LUTs.

ACKNOWLEDGMENT

This work was supported by the Grant Agency of the CTU in Prague, grant No. SGS20/210/OHK3/3T/18 funded by the MEYS of the Czech Republic and the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 co-funded by the MEYS and ERDF

REFERENCES

[1] *UltraScale Architecture-Based FPGAs Memory IP v1.4*, Xilinx, Inc., [cit. 2020-02-01]. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf

[2] "P416 portable switch architecture (psa)," https://p4.org/p4-spec/docs/PSA.html, accessed: 2020-02-05.

[3] P. Benáček, V. Puš, and H. Kubátová, "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.

[4] M. Kekely and J. Korenek, "Mapping of P4 match action tables to FPGA," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–2.

[5] "Barefoot networks unveils tofino™ 2, the next generation of the world's first fully p4-programmable network switch asics: Barefoot," Dec 2018. [Online]. Available: https://www.barefootnetworks.com/press-releases/barefoot-networks-unveils-tofino-2-the-next-generation-of-the-worlds-first-fully-p4-programmable-network-switch-asics/

[6] *Ultrascale+ FPGAs, Product Tables and Product Selection Guide*, Xilinx, Inc., [cit. 2020-02-01]. [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf

[7] *Intel Arria 10 Product Table*, Intel, Inc., [cit. 2020-02-01]. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf

[8] M. Žádník, "Optimization of network flow monitoring," *Information Sciences and Technologies Bulletin of the ACM Slovakia*, vol. 5, no. 1, p. 6, 2013. [Online]. Available: https://www.fit.vut.cz/research/publication/10255

[9] H. Wang, T. Sun, and Q. Yang, "Minimizing area cost of on-chip cache memories by caching address tags," *IEEE Trans. Comput.*, vol. 46, no. 11, p. 1187–1201, Nov. 1997. [Online]. Available: https://doi.org/10.1109/12.644293

[10] G. Barish and K. Obraczke, "World wide web caching: trends and techniques," *EEE Communications Magazine*, vol. 38, no. 5, pp. 178–184, 2000.

[11] K. R. S. Jain, A. Ranjan and A. Raghunathan, "Computing in memory with spin-transfer torque magnetic ram," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 3, pp. 470–483, 2018.

[12] D. Ielmini and H. Wong, "In-memory computing with resistive switching devices," *Nat Electron*, vol. 1, p. 333–343, 2018.

[13] M. L. Gallo and R. M. A. Sebastian, "Mixed-precision in-memory computing," *Nat Electron*, vol. 1, p. 246–253, 2018.

A.5 Paper 5

General Memory Efficient Packet Matching FPGA Architecture for Future High-Speed Networks

Microprocessors and Microsystems 73 (2020) 102950

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

General memory efficient packet matching FPGA architecture for future high-speed networks

Michal Kekely[a], Lukáš Kekely[b,*], Jan Kořenek[c]

[a] Faculty of Information Technology, Brno University of Technology Božetěchova 2, Brno 612 66, Czech Republic
[b] CESNET a. l. e. Zikova 4, Prague 160 00, Czech Republic
[c] IT4Innovations Centre of Excellence Faculty of Information Technology, Brno University of Technology Božetěchova 2, Brno 612 66, Czech Republic

ARTICLE INFO

Article history:
Received 21 March 2019
Revised 3 September 2019
Accepted 6 December 2019
Available online 14 December 2019

Keywords:
FPGA
Packet matching
Packet filtering
High-speed networks
Exact match
Cuckoo hashing

ABSTRACT

Packet classification (matching) is one of the critical operations in networking widely used in many different devices and tasks ranging from switching or routing to a variety of monitoring and security applications like firewall or IDS. To satisfy the ever-growing performance demands of current and future high-speed networks, specially designed hardware accelerated architectures implementing packet classification are necessary. These demands are now growing to such an extent, that in order to keep up with the rising throughputs of network links, the FPGA accelerated architectures are required to perform matching of multiple packets in every single clock cycle. To meet this requirement a simple replication approach can be utilized – instantiate multiple copies of a processing pipeline matching incoming packets in parallel. However, simple replication of pipelines inseparably brings a significant increase in utilization of FPGA resources of all types, which is especially costly for rather scarce on-chip memories used in matching tables.

We propose and examine a unique parallel hardware architecture for hash-based exact match classification of multiple packets in each clock cycle that offers a reduction of memory replication requirements. The core idea of the proposed architecture is to exploit the basic memory organization structure present in all modern FPGAs, where hundreds of individual block or distributed memory tiles are available and can be accessed (addressed) independently. This way, we are able to maintain a rather high throughput of matching multiple packets per clock cycle even without fully replicated memory resources in matching tables. Our results show that the designed approach can use on-chip memory resources very efficiently and even scales exceptionally well with increased capacities of match tables. For example, the proposed architecture is able to achieve a throughput of more than 2 Tbps (over 3 000 Mpps) with an effective capacity of more than 40 000 IPv4 flow records at the cost of only a few hundred block memory tiles (366 BlockRAM for Xilinx or 672 M20K for Intel FPGAs) utilizing only a small fraction of available logic resources (around 68 000 LUTs for Xilinx or 95 000 ALMs for Intel).

1. Introduction

Computer networks and their infrastructure are required to be constantly faster and faster as users want to transfer more data. The ever-increasing capacity of network links leads to a need for all network devices and systems to speed up their packet processing. Even the most powerful current processors are not able to reasonably cope with network traffic on 100 Gbps links. In order to achieve wire-speed processing with a throughput of 100 Gbps and more, network systems have to utilize hardware accelerated FPGA or ASIC technology. The FPGA acceleration provides high performance and is highly configurable (flexible) as well. The flexibility is essential for any practical network system because traffic processing is changing with the introduction of every new protocol, application or service. Therefore, 40 Gbps and 100 Gbps network interface cards with FPGAs (also known as Smart NICs) started to be recently deployed to data centers as hardware platforms for the acceleration [1] and will be probably more and more frequently used in the future.

Flexible network traffic processing can be easily described in the P4 high-level language [2]. Furthermore, this description can be automatically mapped directly to a high-throughput packet processing architecture for an FPGA hardware accelerator [3,4]. The P4

* Corresponding author.
E-mail addresses: ikekelym@fit.vutbr.cz (M. Kekely), kekely@cesnet.cz (L. Kekely), korenek@fit.vutbr.cz (J. Kořenek).

https://doi.org/10.1016/j.micpro.2019.102950
0141-9331/

language has been originally designed at Stanford University in order to enable protocol, vendor and target independent definitions of packet processing. One of the integral parts of the P4 language specification [5,6] is the utilization of match/action tables as a basis to control processing of each input packet.

The core functionality performed by the match/action tables is packet classification in various forms. During the classification process, packets are matched against a set of rules, which are usually defined by exact values, ranges or prefixes of a few selected packet header fields. Generally, the performed classification is a mathematical problem of a multidimensional range search. Due to the large ruleset size and complexity of rules, it is rather difficult to perform matching at such rate that is sufficient for wire-speed processing of high-speed network data. Therefore, many different hardware architectures have been designed to accelerate packet classification [7–13].

To achieve wire-speed 100 Gbps throughput, it is necessary to process every incoming packet only in 6.7 ns, because the shortest 64 B Ethernet packets can arrive within such small time intervals. The time to process a packet corresponds to a 150 MHz clock. It consequently means that multiple packets have to be processed within each clock cycle to achieve wire-speed 400 Gbps or 1 Tbps packet processing if the frequency can not scale over 500 MHz. FPGAs are not ready for such high frequency, especially if large data widths are used to transfer packet data. Therefore, the processing throughput is usually increased simply by the utilization of multiple processing pipelines in parallel [14,15], which require multi-port memories or memory replication. Unfortunately, both approaches significantly reduce throughput scalability at 400 Gbps or 1 Tbps fast links.

Therefore, in this article, we focus on the feasible design of a new hardware acceleration technique for packet classification with efficient utilization of on-chip memory resources to achieve high-speed network traffic processing. We introduce a general novel hardware architecture that is able to scale the throughput of P4 match/action tables to more than 2 Tbps (over 3 000 Mpps) on current FPGAs from both major vendors (Xilinx as well as Intel), while memory replication is significantly reduced compared to other approaches. The proposed concept is compared with a simple pipeline/memory replication scheme and several possible optimizations are introduced. The proposed architecture is further evaluated using various backbone network traffic traces and shown to maintain its high performance even in realistic deployment.

2. Related work

There is a lot of published research in the area of packet classification with many completely different approaches described in individual papers. Some of them focus on being as general as possible, supporting packet classification in multiple different dimensions or supporting different types of match strength such as range lookups, ternary matching or longest prefix match (LPM). However, the only way how to scale most of the published approaches for higher throughputs is to utilize multiple copies of the same architecture operating in parallel. The problem of effective scaling to multiple matches per clock cycle is not properly addressed.

Packet classification based on bit-parallelism (or bit vectors, BV), proposed by Lakshman et al. [16], is a practical implementation that leverages the fact that rule updates are infrequent compared to search operations. The algorithm works in two stages. In the first stage, multiple parallel searches are carried out, each of them limited to only a single (different) dimension of the classification. Each of these parallel searches results in a bit vector that represents which rules were matched (in the given dimension). This means that each bit of these vectors corresponds to one record in classification ruleset, therefore their width is given by the number of rules used. A bit is set to logical one if a corresponding rule is matched in a given dimension and is reset to logical zero otherwise. After this stage, each bit vector represents a subset of rules that were matched in a given dimension. Then the second stage has to find an intersection of the sets matched within single dimensions. Since these sets are represented as bit vectors finding the intersection is reduced to bitwise AND operation among the bit vectors. The main problem with this approach is the width of bit vectors which increases with the number of rules. Song et al. [17] presented architecture that combines bit vector approach with TCAMs. The architecture uses TCAMs for lookups within dimensions that require exact or prefix matches and tree-bitmap implementation of the BV algorithm for source and destination port lookups. This architecture is optimized for classification based on network flow 5-tuples (source IP address, destination IP address, source port, destination port, and L4 protocol), therefore it is not very flexible and was not shown to have the ability to scale to support different header fields.

Several different approaches supporting multiple dimensions are described in [18]. A grid of Tries extends standard Trie to two dimensions however, it is not easily extensible for more dimensions than two. General solution using cross-products is more promising, but with no further optimization uses up way too much memory and resulting cross-products are quite large. Other trie-based algorithms scale poorly with the increasing number of dimensions. Additionally, these algorithms need great amounts of memory and cannot be easily scaled to higher throughputs.

Another group of approaches to classification tries to utilize architectures based on the construction of decision trees. Many of these algorithms are not designed with FPGA implementation in mind, however, some of them can be bent to be efficiently mapped into FPGA structure. HiCuts [8] and HyperCuts [9] are examples of such algorithms. The main idea is to progressively cut the whole searched space represented by classification dimensions into small enough parts (usually representing 1 or only a few rules). Different heuristics can be used to decide how to cut the space efficiently but, resulting trees tend to have many nodes. Additionally adding or removing rules leads to the need for rebuilding of the whole tree.

Prasanna et al. [19] pushed the idea of constructing decision trees even further. They have observed that HyperCuts and similar algorithms do not efficiently deal with rules that have too much overlap with each other. In such cases, many rules need to be duplicated and the resulting tree (hence required memory) can explode exponentially with the number of dimensions. To mitigate this issue, a decision forest is introduced. A ruleset is split into subsets and smaller decision trees are built for each of these subsets. Rules within each subset are chosen so that they have as little overlap as possible and that they specify nearly the same dimensions. Additionally, two other techniques are used to optimize HyperCuts algorithm. Rule overlap reduction stores rules that should be replicated in a list in each internal node instead of actually replicating it into all the child nodes. Precise range cutting is used to determine cutting points which will result in the least number of rule duplications instead of deciding the number of cuts for a field.

Taylor et al. [7] introduced Distributed Crossproducting of Field Labels (DCFL). This algorithm decomposes classification into single dimensions and can be easily parallelized. Moreover, it uses Bloom Filters [11] and labeling technique to lower memory and logic requirements. The architecture was shown to be scalable even to higher throughputs [20], but only by using multiple copies of the memories. Because of these key features, the architecture can be duplicated to increase throughput while still maintaining reasonable usage of on-chip memories and logic. This idea was pushed

further to build scalable architecture through memory duplication in [20].

In many cases, exact match packet classification is sufficient. This is prevalent mainly when IP flows are concerned. Effective approaches to exact match packet classification are usually based on some form of hash tables. A sophisticated way of implementing hash tables is cuckoo hashing principle [21]. The main idea of cuckoo hashing is to increase the efficiency of memory utilization in the hash table by multiple hash functions/tables utilized in parallel. Each table uses one of the different hash functions for indexing its elements. Thanks to this, if a new element cannot be inserted into the first hash table because of a conflict with an already existing item, it can still be inserted into one of the other tables through a different hash function (into a different position). Even when the element cannot be inserted into the correct position in any of the tables it can still be inserted by force, pushing one of the previous occupants out of the tables. The previous occupant can then be reinserted into the tables using the same approach – either finding an empty position in one of the tables or swapping place with another item, which then must be reinserted. The reinsertion process can have multiple iterations with different items. Using more parallel tables and mainly the described reinsertion mechanism allow the cuckoo hashing to keep high lookup speed while decreasing the number of unresolvable conflicts and therefore increasing the effective capacity.

The cuckoo hashing approach is well suited for hardware because each hash table can work in parallel [22,23]. These published implementations offer throughputs up to around only 100 Gbps, while in this paper we aim at achieving over 1 Tbps. Cuckoo hashing based packet classification is also effectively used to monitor or analyze network traffic in the idea of Software Defined Monitoring (SDM) [24]. Here, an external memory is utilized and achieved throughput is again shown to be sufficient only for up to 100 Gbps.

3. Architecture

We aim to design an architecture for exact match packet classification with the main goal being to accommodate very high throughputs in the magnitude of multiple terabits per second. One of the ways to achieve this performance would be to increase the clock frequency of basic cuckoo hashing architectures described at the end of the previous section. However, this is possible to do only until a certain point, after which the frequency cannot be further increased due to the limitations of current FPGA technology. The better way to increased throughput is through the design of a new architecture of cuckoo hashing that can carry out more than one rule lookup per each clock cycle. This would naturally require more than one memory access per clock cycle to each utilized hash table during the matching process. Current FPGAs from both major vendors (Xilinx and Intel) have on-chip tiles of distributed (in logic) and block memories. Block memory tiles are the main type, so we are going to focus on them in the following descriptions and evaluations. However, all of the proposed approaches are general enough to apply to distributed memories as well.

In cases of both vendors, the block memory tiles have two independent read ports, therefore we can easily perform two memory accesses per clock cycle with no replication, and therefore, no additional cost. If we want to enable more than 2 accesses per clock cycle to further increase the throughput, we can simply replicate the memories. An example with 4 accesses is illustrated in Fig. 1. Two of the four accesses are mapped into the first copy of the memory and the remaining two are mapped to the other copy. This approach is not particularly efficient and do not scale well because we need to double the on-chip memory utilization in order to achieve doubled throughput. However, we can leverage the internal structure of FPGAs and their organization of block

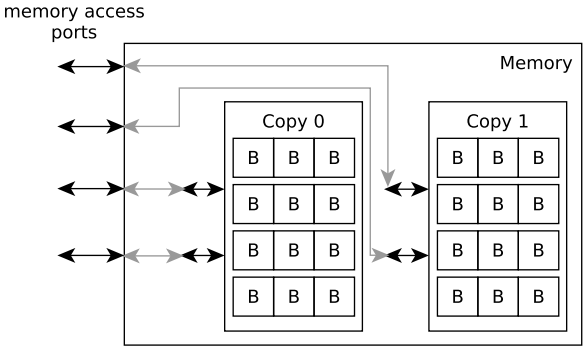

Fig. 1. The memory architecture of simple replication approach using memory tiles on FPGA.

memory into independent tiles. A single copy of a larger memory is internally usually composed of more than one block memory tile (B blocks). More specifically, on current Xilinx FPGA chips each BlockRAM tile [25] can be used as 36 b wide dual port memory with 1 024 entries and on current Intel FPGAs each M20K tile [26] can be similarly used as 20 b wide dual port memory with 1 024 entries. Larger memories are then constructed utilizing multiple BlockRAM or M20K tiles organized into several rows (more entries) and columns (wider data). For example, Fig. 1 corresponds to three columns wide (up to 108 b on Xilinx or 60 b on Intel) and four rows high (up to 4 048 entries) memory.

3.1. Proposed approach

Because there already are multiple rows of memory tiles in each hash table we should be able to perform more than just two memory accesses per clock cycle. In an ideal case, we can, in fact, do two accesses per cycle independently to each of the individual rows of the table. This fact can be leveraged as a key feature to optimize the previously mentioned simple replication approach. Therefore, we propose an FPGA matching architecture of cuckoo hashing shown in Fig. 2. The proposed approach is also easily applicable to any other kinds of hash tables, but we choose cuckoo hashing as it is the most effective existing hashing scheme to date.

An architecture able to carry out up to 2 parallel lookups per cycle with cuckoo hashing using 3 different hash functions/tables is shown in the Fig. 2. The memory blocks used here are similar to the blocks from Fig. 1 – meaning that they internally consist of multiple independent rows of block memory tiles. Hash functions are computed individually for each lookup key (H blocks) and are connected to a distribution logic (D blocks). There is one distributor block for each hash function/table of the cuckoo hashing. The distributor consists primarily of logic that maps the requested memory accesses into corresponding table rows given by a few most significant bits of their hash values (memory address)

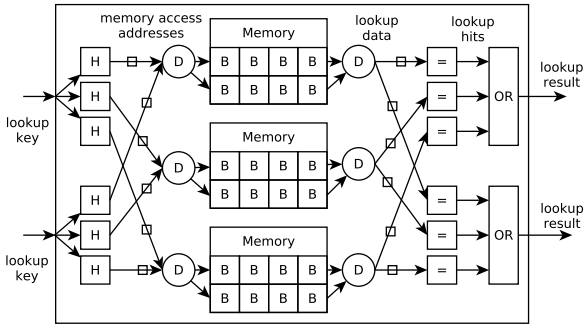

Fig. 2. The top-level architecture of the proposed optimization approach.

and distributes them onto available block memory ports for each of these rows. On the other side of the memory, it correctly forwards data read from each memory row and port to the corresponding comparator logic (= and OR blocks). The basic idea in this concept is to replicate memory fewer times than in the case of a simple approach (fewer replicas than required parallel packet lookups) as we can perform multiple accesses per clock cycle into the table as long as the hash functions are pointing into different rows of memories. Additionally, memory can also be replicated here to enable more than two parallel access ports for each row.

The main function of distributor blocks is to determine which row of the block memory tiles is accessed by which lookup and set the corresponding control logic in a correct way to carry out all the parallel lookups that are not in conflict with one another. Access conflicts, in this case, mean that there are more lookups wanting to access the same row of one table than there are available read ports in this row. Note that since the memories can still be replicated the number of available access ports might be higher than two. All the lookups that could not be carried out in the first cycle will be carried out in consecutive cycles until all of the requested lookups are finished. This means that when access conflicts occur, the lookup of all of the inputs will take more than one cycle. However, the basic idea is that the relative number of occurring conflicts (or rather the number of additional cycles needed) is pretty low, especially for higher numbers of memory rows (larger tables), thus reducing total throughput only very slightly. Compared to that, the saved memory resources thanks to no or weaker replication are considerable.

As an example, let us consider a case where four packet lookups each clock cycle are needed, there are four rows of block memory tiles, and only two access ports per memory (meaning no memory replication). No access conflicts will occur unless at least three of the four parallel lookups need to access the same memory row. In the case of the conflict, two of the conflicting accesses can still be carried out together with all of the others that are not in conflict. The last one or two accesses from the conflicting group has to be carried out in the next clock cycle. Even if there is a conflict every time, we still achieve the same throughput as the simple architecture with the same memory requirements (replication factor). In the example without replication, we would do four lookups in two clock cycles which is the same as the simple approach with two lookups each cycle. This shows that at worst the proposed approach is on par with the simple replication scheme in terms of both memory and throughput. However, the key idea is that the conflicts do not occur each time and are actually pretty infrequent (20% conflict chance in this example), therefore the actually achieved effective throughput is considerably better.

Another important feature of the proposed architecture is that we can easily achieve independence in the access conflict handling for each parallel hash table used in the cuckoo hashing scheme. A distributor corresponding to a single hash table does not need to wait until all the other distributors carried out all their lookups. Instead, there are small input and output buffers that are used to synchronize the access requests and their results (denoted by small squares on corresponding connections in Fig. 2). This makes the architecture a lot more efficient as the throughput is not governed by the probability of no conflicts in all of the tables together but rather by the probability that there are no conflicts in every single table independently. This independent probability is a lot lower especially when a higher number of parallel hash tables are used.

Indeed, the described buffers require some additional FPGA resources. Moreover, the distributors themselves introduce some additional logic overhead compared to simple replication approach. In the simple approach, there is a dedicated memory port for each parallel lookup, therefore hash functions (inputs) and comparison logic (outputs) can be directly connected to appropriate memories without any distributors. The core of each distributor is a simple planner, that can evaluate and resolve access conflicts – basically a group of encoders and decoders to select a valid access plan for each clock cycle. The planner controls two columns of multiplexers: the first to route planed access requests to correct memory rows/ports on the input and the second to pair read data with their corresponding requests on the output. Additional registers are also used to thoroughly pipeline the distributors for better frequency and to correctly synchronize all operations together. The total FPGA logic overhead of the distributors and buffers around them is expected to be manageable compared to complex hashing blocks which are usually considerably large and contain critical paths.

During the resolution of access conflicts, the available access ports of memories are currently not fully utilized in the added clock cycles. For example, if only one lookup cannot be carried out in the first cycle it has to be carried out in the second (additional) one. Reserving one full clock cycle just for one extra lookup is inefficient. A more reasonable approach would be to already combine the extra lookup cycles with some of the lookups needed for the next set of input keys. This approach requires the buffer architecture to be much more complex as it needs to be able to efficiently plan the memory accesses across multiple lookup cycles. This operation is not trivial and would require considerable additional resources and create long critical paths. Furthermore, the resource increase is not exculpable as after our initial experiments we concluded that for the most interesting cases (the ones where the benefits of the proposed architecture are the best) the change would increase the throughput by only a minuscule margin. Therefore the rest of this article does not use architecture with this kind of optimization.

3.2. Analysis of access conflicts occurances

Using some basic probabilities and statistics, it is possible to theoretically analyze the expected probability of access conflict occurrences and thus derive the achievable throughput of the proposed architecture with any given parameters. There are three main parameters of the architecture that influence the probability of conflicts:

1. r as the number of rows of block memory tiles in each table,
2. l as the number of parallel lookups per clock cycle corresponding to the number of inputs,
3. and p as the number of available access ports for each table row.

Using these three parameters we can now examine the conflict probabilities and their effect on the achieved throughput.

The situations when a single lookup needs to access one specific selected row of memory tiles and that it needs to access any other row have mutually complementary probabilities:

$$P_s(r) = \frac{1}{r} \tag{1}$$

$$P_{ns}(r) = \frac{r-1}{r} \tag{2}$$

Now for any given number n, the probability that exactly n lookups out of total l in one cycle need to access one selected row out of r rows can be computed as a product of: the probability that selected n lookups access selected row, the probability that all the other $l-n$ lookups do not access this row, and the number of combinations by which it is possible to position those n colliding

lookups into all l inputs. The corresponding equation is therefore:

$$P_s(n,l,r) = (P_s(r))^n * (P_{ns}(r))^{l-n} * \binom{l}{n} = \left(\frac{1}{r}\right)^n * \left(\frac{r-1}{r}\right)^{l-n} * \binom{l}{n} \quad (3)$$

To get the probability that any of the memory rows will have exactly n lookups mapped onto it we simply multiply the previous probability from Eq. (3) by the total number of rows:

$$P_a(n,l,r) = P_s(n,l,r) * r = \left(\frac{1}{r}\right)^{n-1} * \left(\frac{r-1}{r}\right)^{l-n} * \binom{l}{n} \quad (4)$$

Finally, the probability that more than n lookups out of all l in one cycle need to access the same row out of r can be approximated simply as a sum of the probabilities from Eq. (4) for all values higher than given n:

$$P_{c,a}(n,l,r) = \sum_{i=n+1}^{l} P_a(i,l,r) = \sum_{i=n+1}^{l} \left(\frac{1}{r}\right)^{i-1} * \left(\frac{r-1}{r}\right)^{l-i} * \binom{l}{i} \quad (5)$$

However, this sum does not account for the fact that solution spaces described by some of the summed probabilities can have non-empty intersections with one another (some conflict variants are counted multiple times). To counter this fact we would have to compute probabilities that exactly n lookups will be mapped onto the same row while there is no other row with n or more lookups mapped onto it. This would lead to exponentially more complex nested sums. However, the approximate results achieved by the Eq. (5) are always higher than the actual correct results, which in turn means that they would actually give us more pessimistic results for the throughput. Additionally, this approximation is very precise for results under configurations that are the most interesting for us. For example, it is absolutely precise if p is higher or equal to $l/2$, since in this case, it is impossible for two different rows to have more than p accesses mapped at the same time.

The probability approximated by the Eq. (5) essentially gives the chance that there will be a conflict for a matching architecture with l parallel lookups, r rows of block memory tiles and $p = n$ ports for each row. However, not all occurring access conflicts are equal when it comes to their resolving and thus effect on the total achieved throughput. For example, if $p = 2$ and 6 lookups need to access the same row it takes 3 cycles to carry out all of them, while when only 4 lookups need to access the same row only 2 cycles are needed. To extend our equations and reflect this we introduce weights into the sum:

$$c_{w,c}(n,l,r) = \sum_{i=n+1}^{l} w(i,n) * P_a(i,l,r) \quad (6)$$

The weight w here represents the number of cycles needed to resolve the conflict in each case and can be easily computed as:

$$w(i,n) = \left\lceil \frac{i}{n} \right\rceil \quad (7)$$

Finally, we need to do one last thing in order to get how many times more cycles (on average) are needed compared to the case without any conflicts. The Eq. (6) sums only weighted probabilities of conflicts. We need to also add the probability that there will be no conflict at all. Weight corresponding to no conflict is obviously 1 since even when there is no conflict we still need one clock cycle to carry out all the lookups. So the coefficient that gives us the ration between needed cycles (achieved throughputs) between our architecture and ideal case without conflicts is computed as follows:

$$c(n,l,r) = c_{w,c}(n,l,r) + (1 - P_{c,a}(n,l,r)) \quad (8)$$

In conclusion, the proposed optimized architecture with l lookups, r block memory rows, and p ports can achieve throughput equivalent to an average of m lookups per cycle, where:

$$m = \frac{l}{c(p,l,r)} \quad (9)$$

Thanks to the previously mentioned buffers there is no need to include number of parallel hash functions (hash tables) in the cuckoo hashing scheme into our computations. Lookup processing and memory accesses corresponding to each hash operate independently of one another and their results are only synchronized afterward via buffers. This means that if there is a collision in memory tied to one hash another hash with no collision does not have to wait.

4. Results

Based on the previously described mathematical analysis, the expected throughput results in this section are obtained. Then they are confirmed through extensive experiments with the designed architecture using real network traffic traces. The architecture is implemented in VHDL with configurable parameters like the number of hash tables, their sizes, level of memory replications, and the number of lookups per clock cycle. Measurements of FPGA resources requirements for Xilinx are based on implementations for the UltraScale+ XCVU9P chip [25] using Vivado 2018.2 tool and for Intel are based on implementations for the Stratix10 1SG280HU2F50E1VG chip [27] using Quartus Prime 18.1 Pro. The architecture is able to achieve working frequency ($F_{\max}$) of more than 400 MHz for every evaluated configuration on both chips. Therefore, all throughput results in the following part of this section are shown for 400 MHz clock frequency. We evaluated the architecture for 32 b wide arbitrary data (action) and in two different settings of key width: a 104 b wide key that is sufficient for the classification of standard IPv4 flows (5-tuple), and 296 b wide key for simmilar IPv6 flow matching. There are 3 main parameters that are worth exploring in the obtained results – effective rule capacity, resource requirements (block memory tiles, logic cells), and achievable throughput (lookups per cycle, Mpps, Gbps).

We start the evaluation with achievable capacity analysis of the proposed extended cuckoo hashing scheme. Then continue with resource utilization and scaling for different configurations of the designed atchitecture. And finally, take a closer look at achievable throughputs in real network deployment.

4.1. Effective rule capacity

To examine the effective capacity of the proposed architecture, measurements in several configurations are performed with randomly generated keys. Different key widths (104 b IPv4 and 296 b IPv6 5-tuple), numbers of block memory rows (2, 8, and 16), and numbers of hash functions (2, 3, and 4) are tested. For each configuration, results from 1 000 000 independent runs are obtained, where each run consists of inserting new rules one by one into the cuckoo hash tables until the first unsuccessful attempt occurs. Fig. 3 illustrates the aggregated results of these measurements. The graph shows the probability of achieving at least given rule capacity utilization, e.g. in around 85% of performed tests with two tables (solid line) the achieved utilization is at least 50% of total capacity. More specifically, the depicted results are for IPv4 5-tuples and 8 memory rows (8 192 items) per table. However, changing the key width has no effect on the results at all and changing the number of rows (items) per table has only a negligible effect. The only parameter significantly influencing the effective capacity of the architecture is the number of parallel hash functions – with

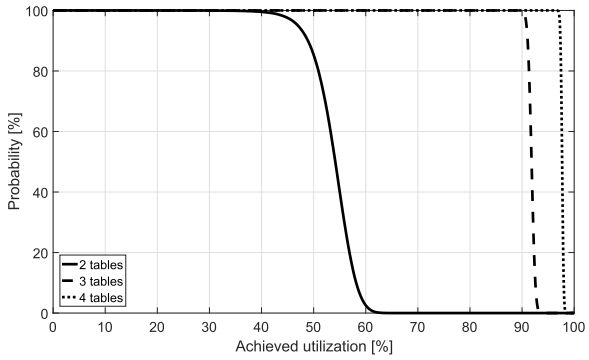

Fig. 3. Achieved memory capacity utilization for a different number of hash tables (functions).

99% probability 40% utilization is achieved with 2 functions, 90% utilization with 3 functions and even 97% with 4 functions.

These results are consistent with similar measurements presented for general cuckoo hashing scheme in published papers like [23]. Therefore, we can conclude that the proposed optimized memory replication scheme has no negative effects on the achievable rule capacity compared to standard cuckoo hashing. Table 1 shows different capacities of the proposed architecture based on the number of hash functions and the number of block memory rows for each table. Total (theoretical) capacity and achievable effective capacity based on the measured results are shown. Table 1 is primarily used to illustrate the capacities of the configurations that are considered in the following evaluation. Note that configurations with two tables are not considered because of the poor achieved utilization.

4.2. Achievable throughput and required resources

The main goal of the proposed approach is to save FPGA memory resources while maintaining high throughput. Therefore, we start the evaluation with Fig. 4 that captures the relations between throughputs and memory tiles utilizations for the designed architecture with three hash functions in different configurations. Each graph shows results for different FPGA vendor (Xilinx in the top half, Intel in the bottom half) and different matching key width (IPv4 5-tuple on the left and IPv6 5-tuple on the right). Outer shape of points (square, circle, triangle) is used to distinguish different numbers of memory rows (last 3 entries in the legend), while different inner shapes of points (middle 3 entries in the legend) are used to represent how many lookups per clock cycle (number of inputs l) the proposed architecture supports. Finally, individual lines in the graphs represent throughput and memory requirements of the simple memory replication approach for a different number of block memory rows used (distinguished by line type from the first 3 entries in the legend). The number of rows is directly tied to the capacity of the architecture as shown in Table 1. These results of the simple replication scheme (lines) form a baseline for evaluation of the designed optimization approach.

Table 1
Capacities of the proposed architecture for different parameters.

Hash functions	Memory rows	Total capacity	Effective capacity
3	1	3 072	2 765
3	2	6 144	5 530
3	4	12 288	11 059
3	8	24 576	22 118
3	16	49 152	44 236
4	1	4 096	3 973
4	2	8 192	7 946
4	4	16 384	15 892
4	8	32 768	31 784
4	16	65 536	63 569

Results of the proposed memory-optimized architecture are shown as individual points in the graphs. The parameters of each evaluated architecture are given by outer and inner shape of the point according to shown legend (e.g. 'x' in a square means 4 rows and 8 lookups). The proposed approach is clearly better in terms of used memory for each given throughput achieved as in each graph all points are below lines that correspond the appropriate number of rows. Obviously, when there is only one row of block memories it is impossible to employ our optimization and gain something. The results for one and two rows of block memories are not shown in the figure for the sake of better clarity. However, even when only two rows of block memories are used we can already achieve better results. For example, using the optimized architecture with 10 lookups we achieve up to 48.5% increase in throughput compared to the simple approach with the same memory requirements (nearly 3 lookups per cycle versus only 2).

When more than two block memory rows per table are used the gains from the proposed approach become even better. With 16 rows (dotted line and triangle points) it is possible to achieve nearly twice the throughput without any memory duplication even when using the proposed architecture with only 4 lookups (cross). If we use versions with 8 ('x') or 10 (star) lookups per cycle the speedup is even further amplified and nearly 7 or 7.5 times higher throughput is reached with no additional memory requirements. Additionally, when utilizing two replicas of memory, the proposed approach can achieve nearly the full throughput of 10 lookups per cycle. In other words, we achieve 99.7% of throughput with only 40% of used memory compared to simple replication. The observed increase of speedup for architectures with more rows in their tables is expected. Because more rows mean a higher chance for the lookups to be better spread out between different rows and thus the probability of access conflicts occurring during matching decreases.

Comparing individual graphs in Fig. 4 to one another, we notice that they all look very similar. The only real difference is the scale of their y-axes (memory requirements), while all the general characteristics described above remain the same. Therefore, the memory savings offered by the proposed architecture scale comparably well regardless of key width and FPGA vendor. Furthermore, the total required memory seems to scale linearly with configured key width as 2.5 $\times$ increase in memory utilization is evident between architectures with 104 b wide IPv4 flow identifier (left half) and 296 b wide IPv6 flow identifier (right half). Finally, visible nearly 2 $\times$ increase in the number of utilized memory tiles between Intel (bottom) and Xilinx (top) is due to different sizes of one tile between the two FPGAs – Intel M20K tile is smaller at 20 Kb (20 b $\times$ 1024 items) while Xilinx BlockRAM tile has 36 Kb (36 b $\times$ 1024 items). The actual size of the utilized memory in bytes is, therefore indeed, comparable on both devices.

The efficiency of the proposed memory optimization approach is not affected by the number of used hash functions, they only affect the effective rule capacity. This can be clearly seen by comparing Fig. 5 with the appropriate graph from Fig. 4. Fig. 5 shows the relation of utilized memory and achieved throughput for different architecture configurations with 4 hash functions on Xilinx FPGAs using IPv4 flow identifiers. Graphs for Intel FPGAs and IPv6 flow identifiers are now omitted as they are again nearly identical to one another. Graphs shown by Figs. 4 and 5 are pretty much the same only shifted slightly along the y-axis. This increase in memory requirements is offset by the higher capacity of the architectures with 4 hash functions (see Table 1).

Results about memory requirements presented so far might suggest that architectures with more lookups per cycle (inputs) are

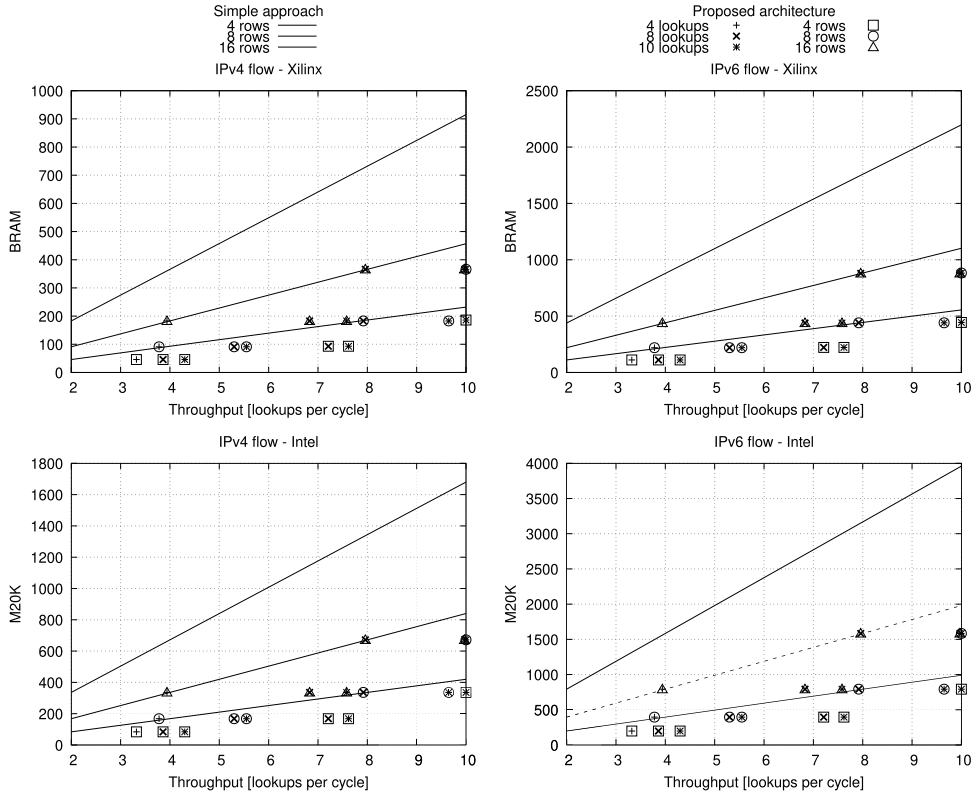

Fig. 4. The relations between utilized memory tiles and achieved throughput for different FPGAs and key widths when using 3 hash functions.

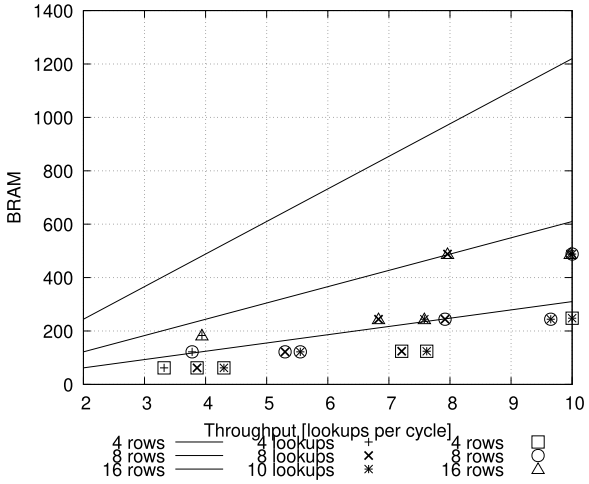

Fig. 5. The relation between utilized memory and achieved throughput for Xilinx FPGAs and IPv4 flows when using 4 hash functions.

always better. However, this is not the case when it comes to utilized on-chip logic resources. Each additional lookup port needs its own hash function implementation, independent buffers, and generally leads to larger distributors. The relation between utilized on-chip logic, more specifically required LUTs or ALMs, and throughput for 3 hash functions is illustrated in Fig. 6. Each graph shows the results for different FPGA vendor and different matching key width and the relations are again similar. We can see that if we use an architecture with for example 10 lookups (star) the logic requirements go up together with the level of memory duplication and the achieved throughput. Memory-optimized architecture with 10 lookups, 16 rows and 4 memory ports (two memory replicas) achieves 99.7% of throughput requiring only 40% of memory at a cost of around 466% of LUTs compared to the simple approach with 10 lookups and 16 rows regardless of device and key width. From a different point of view, the optimized architecture with 10 lookups, 2 rows and 2 memory ports (no replication) achieves 48.5% increased throughput requiring the same memory at a cost of at most 244% increase in LUTs/ALMs compared to the simple approach with 2 lookups and 2 rows. However, we argue that the decreased memory requirements or increased throughput, depending on the way we look at it, is a favorable trade-off for the increase in on-chip logic. In many cases, even the increased logic requirements are still feasible for current FPGAs (only a few percents of the total available), while increasing the throughput without the need to replicate memories can prove to be more critical.

Finally, let's analyze the proposed memory optimized architecture from a different point of view. What is the best achievable result if we want to reach a given throughput goal? Figs. 7 and 8 illustrate memory requirements of the best configurations of the proposed approach (best proposed) compared to the baseline given by the simple memory replication (simple) when reaching a throughput of at least 800 Gbps or 2.4 Tbps respectively. The best configuration is the one that requires the least memory while still satisfying the minimal throughput threshold. This obviously means that actually achieved throughputs of compared simple and optimized configurations are not the same. For better comparison, we can leverage the fact that memory of the simple approach scales linearly with throughput and adjust the required memory to the point where the simple approach has exactly the same throughput as the optimized (adjusted simple). We can see that the proposed approach becomes more and more effective as the total capacity of the cuckoo hash table (number of rows) rises. For 2 rows it is possible to achieve the same throughput as simple replication with somewhere between 67% and 80% of required memory (after adjustment), while for 16 rows only between 25% and 40% of

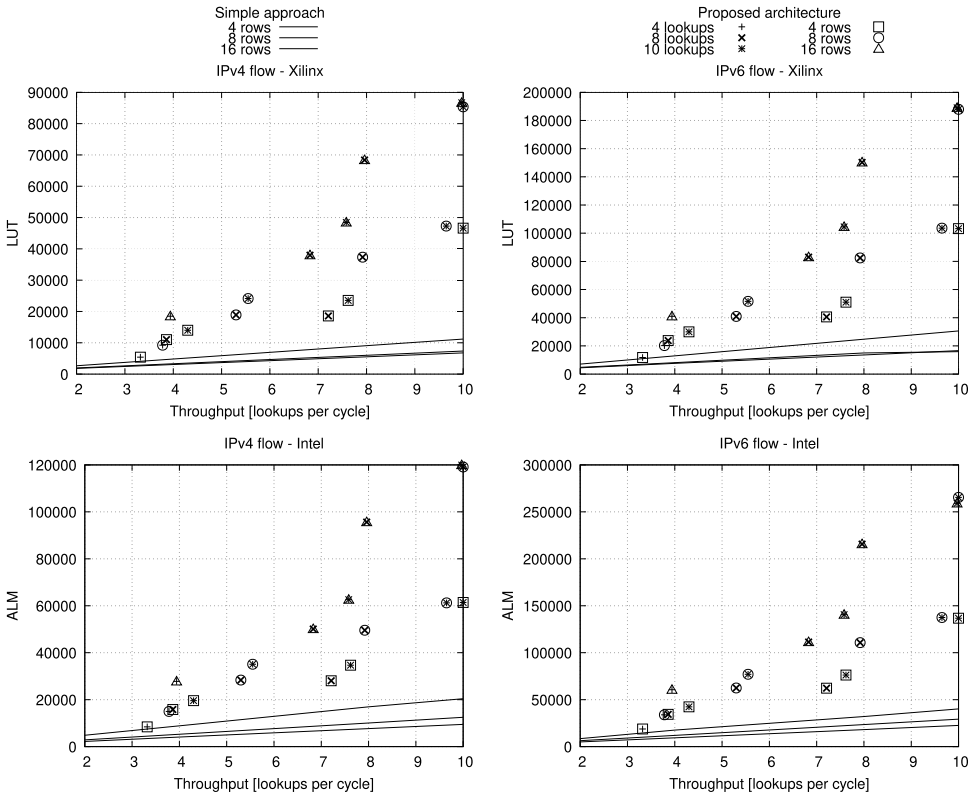

Fig. 6. The relation between utilized logic and achieved throughput for different FPGAs and key widths when using 3 hash functions.

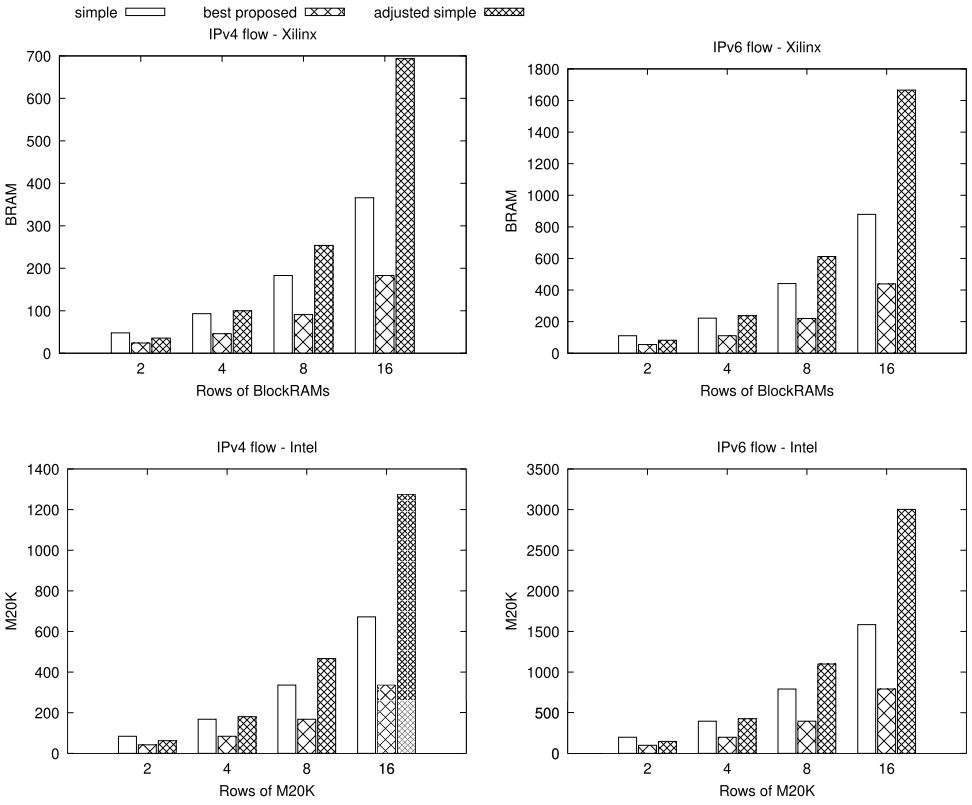

Fig. 7. Memory requirements comparison when achieving at least 800 Gbps.

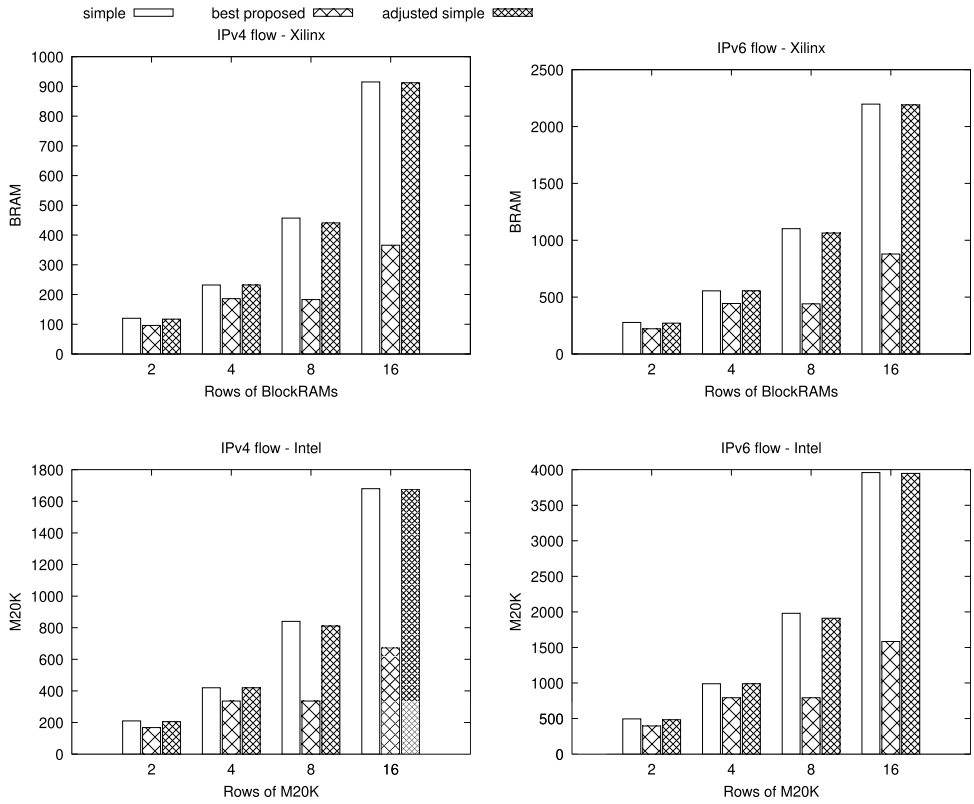

Fig. 8. Memory requirements comparison when achieving at least 2.4 Tbps.

memory is needed. To be more precise the most significant factor that governs memory savings is the ratio between the number of rows (capacity) and required throughput (parallel lookups). The higher the capacity the better the results become as the lookups can be spread among more rows. This is true regardless of key width and FPGA vendor.

4.3. Evaluation on real network traffic

The previous results are obtained under a premise that the network traffic, or more specifically packets and their identifiers used in the matching, have random and evenly distributed values. However, this is not always the case in real networks. Most likely there is a multitude of ongoing sessions between different devices each composed of multiple packets that are transferred in short bursts. The burstiness of the traffic from the same flow means that the probability of multiple packets accessing the same row of block memories might be higher than what we obtained through mathematical analysis. To better understand performance limits of the proposed approach we, therefore, analyze achieved throughput on real network traces. We mainly focus on two scenarios.

The first examined scenario uses only packet identifiers extracted from the network traces and assumes that each packet has the shortest possible length of 64 B. Here, only the distribution of the matching identifier is taken into account and therefore, only basic patterns found in the real network traces are demonstrated. For the matching architecture, this is the worst-case scenario under the full network load, because the maximal possible amount of packets needs to be classified in each and every clock cycle. In the second scenario, we also take into account another characteristic of the network traffic – actual packet length. Longer packets mean that within the same time window (one clock cycle) fewer packets need to be actually classified and therefore there is a lower chance of access conflict occurring. This represents the average-case on a real network more closely. However, the exact timing of packet arrivals is still ignored to simulate full network load.

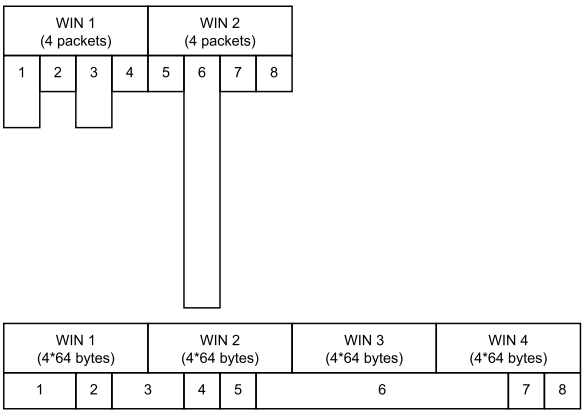

Fig. 9. Illustrations of examined scenarios with worst-case packet window and average-case byte window.

Both scenarios are better illustrated by Fig. 9. In the worst-case scenario (top part), there is a window of l packets processed in every clock cycle because we assume each packet to be of a minimal length of 64 B. Where l is the number of parallel lookups supported by the architecture. In the average-case scenario (bottom part), we instead have a window of l*64 bytes as the amount of data received for processing in each clock cycle. If a packet does not fully fit into a single window and is spread among multiple, it will be classified in the last window it occupies. The architecture still needs to classify all of the packets ending in a single window each clock cycle or add additional cycles if access conflicts occur. For example in the worst-case scenario, packets 1 to 4 arrive in the first window (first clock cycle), packets 5 to 8 in the second,

Table 2
Basic characteristics of capture traffic traces from CESNET network.

Trace name	Packets	Bytes	Time period	Capture time
meter1	1 000 000	1 081 259 293	1.033 s	11:00
meter4	1 000 000	791 590 133	1.489 s	15:00

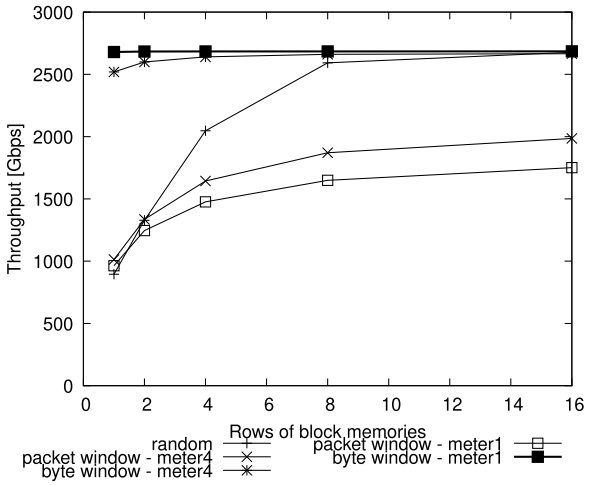

Fig. 10. Throughput results on real network traces for architecture with 10 lookups and 2 × memory replication.

and so on. In the average-case scenario, packets 1 to 2 arrive in the first window, packets 3 to 5 in the second, no packets in the third window, and packets 6 to 8 in the last window.

Real network traces used for this evaluation were obtained from the high-speed backbone network managed by CESNET. CESNET is Czech National Research and Educational Network which has optical links operating at speeds of up to 100 Gbps. This optical network serves around 200 000 users and routes mainly IP traffic. Data traces were captured at different points of the network and at different times of the day. The captured traces contain both IPv4 and IPv6 flows, with IPv4 dominating. To support matching of both IP versions, results for architectures supporting IPv6 5-tuples are shown and IPv4 addresses in rules are extended to apropriate width. In the following evaluations two traces are used: *meter1* and *meter4*. Their basic characteristics are shown in Table 2.

Basic look at the achievable throughput under realistic deployment is provided in Fig. 10. It shows achieved throughput for different numbers of block memory rows on captured network traces. A reference throughput for packets with random uniformly distributed identifiers (results from Section 4.2) is shown as well as measured results using packet and byte windows. On both network traces, the architecture shows similar behavior. The main things we can observe is that for the worst-case scenario (packet window) the throughput is overall lower than in random case and for the more realistic average-case (byte window) it is always higher than expected. The interesting fact to notice about the worst-case scenario is that its throughput falls behind the expected values more and more with the rising number of block memory rows. This behavior would suggest a more prevalent occurrence of access collisions than expected. However, if we take into account realistic packet lengths (byte window) the observed decrease in throughput is far outweighed by the lower arrival rate of matching requests into the architecture.

Fig. 11 shows that the same trends of achieved worst-case and average-case throughputs can be observed even when we decrease the level of memory replication. This graph shows a comparison between the same 2 × replication as used in Fig. 10 and architecture with no memory replication at all. With decreased replication levels the results just get shifted a bit towards lower throughputs, but the observed trends remain the same. Furthermore, if we change the number of lookups that the architecture can at most perform per clock cycle similar trends in the graph arise again. Fig. 12 shows a comparison of measured throughputs for two architectures that differ only in the number of lookups per clock cycle. Once again the random data measurement has a higher throughput than the worst-case scenario, but average-case still beats the expectations. Finally, a somehow different picture arises if we try to classify the captured packets based only on source IP address and not the whole IP 5-tuple. The measured results are provided in Fig. 13. We can see, that the decrease in throughput for the worst-case scenario is slightly bigger for IP address only matching than for IP flows matching. This, in turn, would suggest a higher probability of access conflicts occurrence for less specific packet identification.

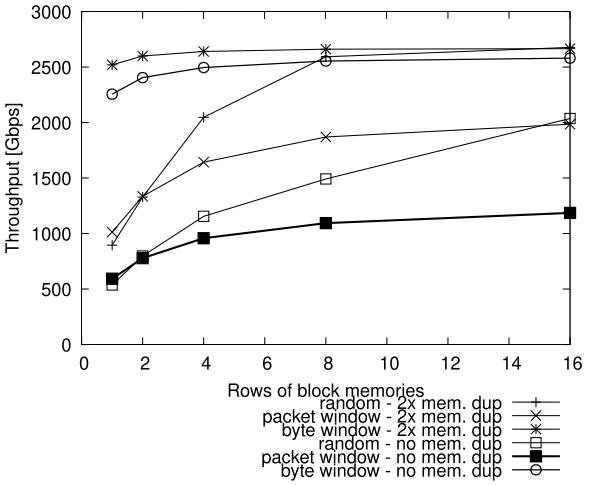

Fig. 11. Throughput results on real network trace *meter4* for architecture with 10 lookups and different levels of memory replication.

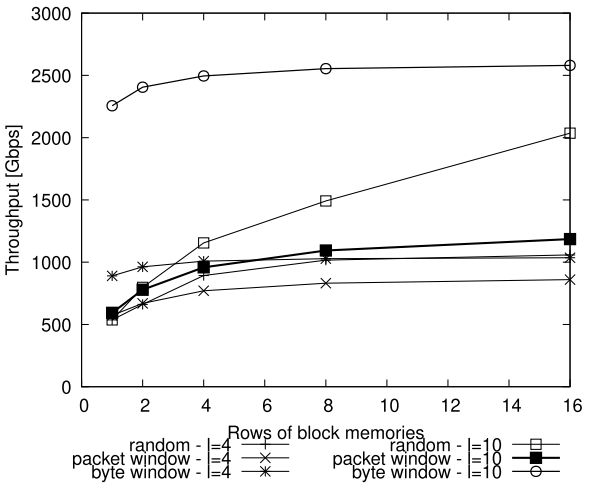

Fig. 12. Throughput results on real network trace *meter4* for architecture with different numbers of lookups.

The above-described reduction of throughput in the worst-case scenario can be easily explained after further analysis of the occurring access conflicts. In the real network data, the majority of the conflicts are caused by multiple subsequent packets with exactly the same identifiers. In evaluated cases belonging to the same flow or originating from the same network device. The probability of this special type of collision does not decrease when we increase the number of block memory rows, because these packets

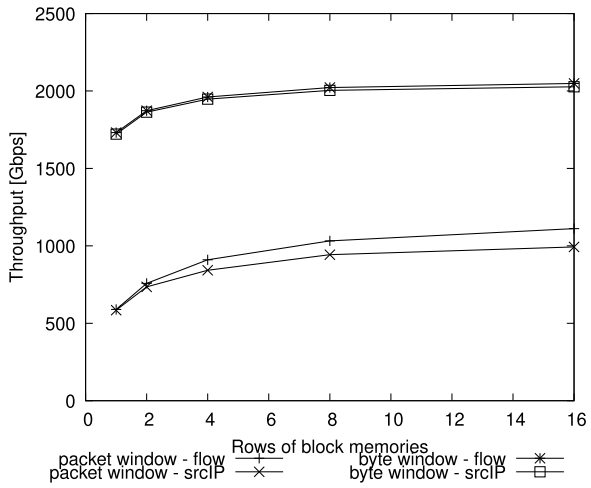

Fig. 13. Throughput results for architecture with 8 lookups and no memory replication with different keys.

need to access not only the same row but exactly the same item (entry) in each table. However, this special case does not actually have to cause an access conflict and stall the pipeline. Although multiple lookups need to access the same row, they require exactly the same entry from the memory. Therefore, this entry needs to be only read out once and then distributed as a result for all of the lookups that needed it. This leads to a possible optimization, where additional logic (simple address comparators) can be added into each distributor that handles these types of parallel lookups and aggregates them into only one memory access. With this optimization even the worst-case scenario throughput exactly matches the expected results from Sections 3.2 and 4.2. This change does not pose any significant increase in overall resources and benefits the throughput only under a very specific traffic pattern in the worst-case scenario. Therefore the actual results for logic consumption are left out of this article.

5. Conclusion

The article presents and examines the design of a novel memory optimized FPGA architecture for general exact match packet classification at very high speeds (400 Gbps and beyond) based on the cuckoo hashing algorithm. The proposed architecture offers an easily configurable tradeoff between total achieved throughput, required on-chip memory, utilized logic resources, and effective rule capacity with equally favorable scaling on FPGAs from both major vendors. Thanks to the designed unique memory optimization scheme, it is possible to implement exact match packet classification at very high throughputs even for large rulesets operating with efficient utilization of available memory. There are several ways in which the architecture can be effectively used – either to maximize throughput and rule capacity on devices with limited memory resources or to minimize memory requirements while satisfying needed rule capacity and throughput.

Thorough experimental measurements of the proposed simple and optimized architectures of cuckoo hashing presented in the article show several interesting facts. First of all the proposed memory optimized architecture is considerably more efficient than a simple replication approach presented in the related works while it still retains exactly the same effective rule capacity as the original cuckoo hashing approach. For appropriate configurations, we are able to achieve up to 99.7% of the original throughput for only 25 to 40% of utilized memory resources compared to the simple replication. The achieved memory savings gets higher when hash tables with larger capacities are used. Thanks to this favorable scaling we can achieve an unprecedented throughput of 2.4 Tbps with an effective capacity of over 44 000 IPv4 5-tuple (flows) rules when using on-chip block memories for the cost of only 366 BlockRAM tiles on Xilinx FPGAs or 672 M20K tiles on Intel FPGAs. Similarly, even a feasible IPv6 5-tuple matching can be implemented with the same throughput and capacity at the cost of 882 BlockRAMs or 1 584 M20Ks. The only downside of the proposed memory optimized architecture is the increased requirement of on-chip logic resources. However, the increase is well within manageable margins and we argue that the benefits of decreased memory requirements and increased throughput outweigh this issue in most practical cases. Finally, the performance of the architecture is proven to hold (after the proposed same item access optimization) when processing real network traffic even in the worst-case scenario when flooded by the shortest packets and for the average-case, the total throughput is even higher than expected.

Declaration of Competing Interest

We are not aware of any existing conflicts of interest.

Acknowledgments

This research is supported by the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 by the MEYS of the Czech Republic; the IT4Innovations excellence in science project IT4I XS – LQ1602; and by the Ministry of the Interior of the Czech Republic projects VI20172020064 and VI20152019001.

References

[1] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, D. Burger, A cloud-scale acceleration architecture, in: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2016.
[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker, P4: programming protocol-independent packet processors, SIGCOMM Comput. Commun. Rev. 44 (3) (2014) 87–95.
[3] P. Benáček, V. Puš, H. Kubátová, P4-to-VHDL: automatic generation of 100 Gbps packet parsers, in: Proceedings of the IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2016, pp. 148–155.
[4] P. Benáček, V. Puš, H. Kubátová, T. Čejka, P4-To-VHDL: automatic generation of high-speed input and output network blocks, Microprocessors and Microsystems 56 (2018) 22–33.
[5] The P4 Language Consortium, The P4 Language Specification: Version 1.0.5, 2018.
[6] The P4 Language Consortium, $P4_{16}$ Language Specification: Version 1.1.0, 2018.
[7] D. Taylor, J. Turner, Scalable packet classification using distributed crossproducing of field labels, in: Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 2005, pp. 269–280.
[8] P. Gupta, N. McKeown, Packet classification using hierarchical intelligent cuttings, in: Proceedings of the Hot Interconnects, 1999.
[9] S. Singh, F. Baboescu, G. Varghese, J. Wang, Packet classification using multidimensional cutting, in: Proceedings of the Conference on Applications, Technologies, architectures, and Protocols for Computer Communications, ACM, New York, NY, USA, 2003, pp. 213–224.
[10] H. Lee, W. Jiang, V.K. Prasanna, Scalable high-throughput SRAM-based architecture for IP lookup using FPGA, in: Proceedings of the International Conference on Field Programmable Logic and Applications, 2008.
[11] S. Dharmapurikar, H. Song, J. Turner, J. Lockwood, Fast packet classification using Bloom filters, in: Proceedings of the 2006 ACM/IEEE symposium on Architecture for Networking and Communications Systems, ANCS, ACM, New York, NY, USA, 2006, pp. 61–70.
[12] V. Puš, J. Kořenek, Fast and scalable packet classification using perfect hash functions, in: Proceedings of the 17th International ACM/SIGDA Symposium on Field Programmable Gate Arrays, FPGA, ACM, New York, NY, USA, 2009.
[13] J. Kořenek, V. Puš, J. Blaho, Memory optimization for packet classification algorithms, in: Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, in: Association for Computing Machinery, Association for Computing Machinery, 2009, pp. 165–166.
[14] H. Le, V.K. Prasanna, Scalable Tree-based Architectures for IPv4/v6 Lookup Using Prefix Partitioning 61 (7) (2012) 1026–1039. ISSN 0018-9340

[15] Y. Qi, J. Fong, W. Jiang, B. Xu, J. Li, V. Prasanna, Multi-dimensional packet classification on FPGA: 100 GBPS and beyond, in: Proceedings of the International Conference on Field-Programmable Technology
[16] T.V. Lakshman, D. Stiliadis, High-speed policy-based packet forwarding using efficient multi-dimensional range matching, SIGCOMM Comput. Commun. Rev. 28 (4) (1998) 203–214.
[17] H. Song, J.W. Lockwood, Efficient packet classification for network intrusion detection using FPGA, in: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays, FPGA, ACM, New York, NY, USA, 2005, pp. 238–245.
[18] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, Fast and scalable layer four switching, SIGCOMM Comput. Commun. Rev. 28 (4) (1998) 191–202.
[19] W. Jiang, V.K. Prasanna, Scalable packet classification on FPGA, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 20 (2012).
[20] M. Kekely, J. Korenek, Packet classification with limited memory resources, in: Proceedings of the Euromicro Conference on Digital System Design, Institute of Electrical and Electronics Engineers, 2017, pp. 179–183.
[21] R. Pagh, F.F. Rodler, Cuckoo hashing, in: Algorithms - ESA 2001, volume 2161 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 121–133.
[22] A. Kirsch, M. Mitzenmacher, Y. Baohua, X. Yibo, L. Jun, Using a queue to de-amortize cuckoo hashing in hardware, http://www.eecs.harvard.edu/michaelm/postscripts/aller2007.pdf 2007.
[23] L. Kekely, M. Žádník, J. Matoušek, J. Kořenek, Fast lookup for dynamic packet filtering in FPGA, in: Proceedings of the 17th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, IEEE Computer Society, Warszaw, Poland, 2014, pp. 219–222. ISBN: 978-1-4799-4558-0.
[24] L. Kekely, J. Kucera, V. Pus, J. Korenek, A.V. Vasilakos, Software defined monitoring of application protocols, IEEE Trans. Comput. 65 (2) (2016) 615–626.
[25] Xilinx, UltraScale and UltraScale+ FPGAs Packaging and Pinouts, Xilinx Inc., 2016. UG575.
[26] Intel, Intel Stratix 10 Embedded Memory User Guide, Intel Corporation, 2018. UG-S10MEMORY 2018.12.24.
[27] Intel, Intel Stratix 10 GX/SX Device Overview, Intel Corporation, 2019. S10-OVERVIEW 2019.02.15.

Michal Kekely is a Ph.D. student at Faculty of Information Technology, Brno University of Technology since 2016 and also an FPGA firmware developer at the Research and development department of Netcope Technologies since 2016. Michal's research is focused mainly on hardware accelerated solutions for high-speed networks, particularly in the area of network monitoring and security. So far, he is an author of several research papers published at renowned international conferences.

Lukáš Kekely received his Ph.D. degree from Faculty of Information Technology, Brno University of Technology in 2017. Lukáš is a researcher and a project manager at the hardware department of Liberouter project which is a part of CESNET (Czech National Research and Educational Network). The main focus of his research is the hardware acceleration of time-critical networking operations using FPGAs, particularly in the area of high-speed network security and monitoring. He is an author of many research papers published at renowned international conferences.

Jan Kořenek received his Ph.D. degree from Faculty of Information Technology, Brno University of Technology in 2010 and recently finished his inaugural dissertation there. He is an assistant professor at the Brno University of Technology and a head of the Security and administration tools department (Liberouter project) of CESNET. He has substantial experiences in the hardware acceleration of network applications which was obtained especially by working on a number of European and locally funded research projects. He is an author of many conference papers, journal articles, and novel hardware architectures. In May 2007, he co-founded INVEA-TECH (now separated into Netcope Technologies and Flowmon Networks) which is an internationally successful spin-off focused on high-speed network monitoring and security applications. In 2009, he also formed Accelerated Network Technologies (ANT) research group at Brno University of Technology.

A.6 Paper 6

Optimizing Packet Classification on FPGA

2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)

Optimizing Packet Classification on FPGA

Michal Kekely
FIT BUT
Božetěchova 2, 612 66 Brno
Czech Republic
ikekelym@fit.vutbr.cz

Jan Kořenek
FIT BUT
Božetěchova 2, 612 66 Brno
Czech Republic
korenek@fit.vutbr.cz

***Abstract*—Packet classification is a crucial time-critical operation for many different networking tasks ranging from switching or routing to monitoring and security devices like firewalls or IDS. Accelerated architectures implementing packet classification must satisfy the ever-growing demand for current high-speed networks. However, packet classification is generally used together with other packet processing algorithms, which decreases the available hardware resources on the FPGA chip. The introduction of the P4 language requires the packet classification to be even more flexible while maintaining a high throughput with limited resources. Thus, we need flexible and high-performance architectures to balance processing speed and hardware resources for specific types of rules. DCFL algorithm provides high performance and flexibility. Therefore, we propose optimizations to the DCFL algorithm and overall packet processing hardware architecture. The goal is to maximize the throughput and minimize the resource strain. The main idea of the approach is to analyze the ruleset, identify some conflicting rules and offload these rules to other hardware modules. This approach allows us to process packets faster, even in the worst-case scenarios. Moreover, we can fit more packet processing into the FPGA and fine-tune the packet processing architecture to meet a specific network application's throughput and resource demands. With the proposed optimizations we can achieve up to a 76 % increase in the throughput of the packet classification. Alternatively, we can achieve up to a 37 % decrease in resources needed.**

I. Introduction

The capacity and throughput of network links are steadily growing. Network traffic processing needs to keep up with this growth, which leads to the need for faster packet processing algorithms. For 400 Gbps networks, it is necessary to process one packet in 1.25 ns. We need to have appropriate hardware acceleration methods to achieve these processing speeds.

Using FPGA and ASIC provides high performance that helps achieving the speeds required by today's networks. However, flexibility is also essential for any practical network system because traffic processing changes with the introduction of every new protocol, application or service. Therefore, network interface cards with FPGAs are being deployed to data centres as hardware platforms for the acceleration [1].

Moreover, flexibility is required by introducing high-level languages for packet processing, such as the P4 language [2]. The P4 language has been designed to enable protocol, vendor and target independent definitions of packet processing. Due to protocol independence, the language has widespread use in different network applications. These applications also introduce a wide spread of requirements on throughput, rule capacity and even the types of matches and their complexity. An integral part of the P4 language is utilizing Match/Action tables (packet classification) to control processing of each input packet. The system described in P4 language can be directly mapped into FPGA [3], [4].

The rulesets used can be big and complex, which introduces even more strain on the resources and performance of the packet classification algorithms and architectures. Additionally, real use-cases consist of multiple different lookups that need to be carried out for each packet (for example L2 filtering followed by L3 forwarding and so on). Therefore, making the packet classification as efficient as possible is crucial.

The main goal of this paper is to optimize packet classification based on the DCFL algorithm [5] to achieve higher throughputs and lower resource consumption. While DCFL is a flexible and fast approach, its throughput relies on the complexity of the ruleset. Therefore, the throughput is decreased for specific complex rulesets. We propose new optimization techniques which provide decomposition of the ruleset to separate modules based on the offline or online analysis. The analysis aims to determine the most interfering and complex rules that decrease the performance of the DCFL algorithm. These rules are then removed from the ruleset and handeled separately. This optimization can increase throughput of the standard 5-tuple packet classification by up to 76 % or can decrease hardware resources by up to 37 %.

The paper outlines related work in section II. Afterwards, the theory behind the optimizations is shown in section III. Results achieved by the optimizations for different rulesets are compared to the baseline DCFL and other approaches in section IV.

II. Related Work

The most general and flexible approach to packet classification is to use ternary content-addressable memory (TCAM). Data stored in these memories can contain don't care or wildcard bits, whose values are ignored during the lookup. Therefore, the approach can be used to match direct values as well as prefixes, masked values, and ranges. During lookup, every entry in the TCAM is checked in parallel to achieve high throughput. However, this means that TCAM does not scale

979-8-3503-3277-3/23/$31.00 ©2023 IEEE

well when increasing its capacity. The complexity of the memory goes up exponentially.

A more efficient way of scaling is to use algorithmic TCAM. These approaches use smaller TCAMs combined with different approaches or with some additional logic that efficiently uses available FPGA resources - for example, slice registers [6], SRAMs [7], or LUTRAMs [8]. Song et al. [9] presented an architecture that combines bit vector approach with TCAMs. This architecture is optimized for classification based on network flow 5-tuples, therefore it is not very flexible and was not shown to have the ability to scale to support different header fields.

Another group of approaches to classification tries to utilize architectures based on the construction of decision trees. Many of these algorithms are not designed with FPGA implementation in mind, but, some of them can be bent to be efficiently mapped into FPGA structure. HiCuts [10] and HyperCuts [11] are typical examples. The main idea is to cut the whole searched space represented by classification dimensions into small enough parts (usually representing 1 or several rules). Different heuristics can be used to decide how to cut the space. But, resulting trees tend to have many nodes. Additionally, adding or removing rules leads to the need of rebuilding the whole tree. Several different approaches to support multiple dimensions (matching on multiple packet header fields) are described in [12]. A grid of tries extends standard trie structure to two dimensions, but it is not easily extendable to more than two. Other trie-based algorithms scale poorly with increasing number of dimensions. Additionally, these algorithms need large amounts of memory and cannot be easily scaled to higher throughputs.

Prasanna et al. [13] pushed the idea of constructing decision trees even further. The authors have observed that HyperCuts and similar algorithms do not efficiently deal with rules that have too much overlap with each other. In such cases, many rules need to be duplicated and the resulting tree (and required memory) can explode exponentially with the number of dimensions. To reduce the exponential growth, a decision forest is introduced. Ruleset is split into subsets and smaller decision trees are built for each subset.

In many cases, exact match packet classification is sufficient. This is prevalent mainly when IP flows are concerned, for example, to filter network traffic (blacklisting specific communications). Effective approaches to exact match packet classification are usually based on hash tables. One way of implementing hash tables is cuckoo hashing principle [14]. The main idea of cuckoo hashing is to increase the efficiency of memory utilization in the hash table by multiple parallel hash functions and tables. Each table uses one of the different hash functions for indexing. Even when the element cannot be inserted into any of the tables it can still be inserted by force, pushing out one of the previous occupants which is then reinserted. This allows the cuckoo hashing to keep the high lookup speed while decreasing the number of unresolvable conflicts and therefore increasing the effective capacity. The cuckoo hashing approach is also well suited for hardware implementation as each hash table can work in parallel [15], [16]. The approach was also shown to be scalable to higher throughputs [17].

Taylor et al. [5] introduced Distributed Crossproducting of Field Labels (DCFL). This algorithm decomposes classification into single dimensions and can be easily parallelized. Moreover, it uses Bloom Filters [18] and labeling technique to lower memory and logic requirements. The architecture was shown to be scalable even to higher throughputs [19], but only by using multiple copies of the memories which increases the memory consumption and might decrease rule capacity. The architecture is fast, flexible, scalable and has good hardware implementation and therefore is suitable for many different network applications running in FPGA. However, throughput of the approach is given by the size of crossproducts between different dimensions when combining them and as such is highly dependent on the actual ruleset. For larger and more complex rulesets the algorithm might struggle to meet the throughput requirements without duplicating large amounts of memory. Therefore, in this paper, we focus on alleviating this downside of the DCFL algorithm by decomposing the rulesets.

III. DCFL Analysis

This paper aims to optimize DCFL architecture for mapping the description in the P4 language to the FPGA. All of the optimizations are focused on flexibility and scalability. We want to achieve desired throughputs by using as few hardware resources as possible.

An example of DCFL-based architecture is illustrated for 4-dimensional classification in figure 1. The architecture works as a pipeline. First of all single-dimensional lookups are carried out in units *D1*, *D2*, *D3* and *D4* (this can be done using any single-dimensional algorithm, based on the type of match required in the given dimension). Afterwards, the results from each dimension are aggregated through a series of aggregation units. First unit aggragates two sets of results (R^1 and R^2) from two dimensions (*D1* and *D2*) into one new set of possible results $R^{1,2}$ by doing a crossproduct of $R^1 x R^2$ and then finding a subset of the combinations that are actually contained within an existing rule. Following units then aggregate this submatch for a combination of dimensions $R^{1,\ldots,i}$ with the match for another dimension R^{i+1} into a new submatch $R^{1,\ldots,i+1}$. Bloom filter arrays and meta-label indexing is used to do this. Essentially for each member of the crossproduct a membership to the resulting set needs to be determined.

DCFL uses decomposition to increase processing speed and provide more flexibility and scalability. However, two possible bottlenecks of this architecture are the single-dimensional lookups and the aggregation units, which have their throughput tied to the sizes of the crossproducts that are created. For larger and more complex rulesets the sizes of the crosspruducts increase and the aggregation units need to be duplicated to achieve desired throughput.

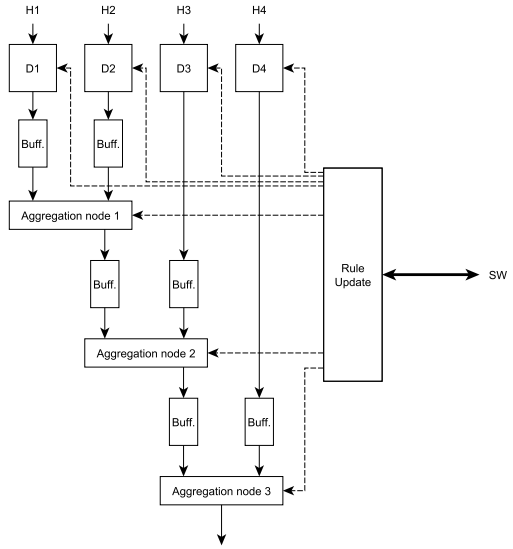

Fig. 1. Top-level architecture of DCFL algorithm.

A. Single-dimensional lookups

Implementation of the single-dimensional lookups is dependent on the type of matching that needs to be done in each dimension. For exact match the most efficient approach is to use hashes, more specifically cuckoo-hash-based approach [16]. For more complex matching (ternary, longest prefix match) trie-based solutions [10] and TCAMs can be used. Most of these single-dimensional approaches have already been shown to be scalable and optimized - Cuckoo hashing by clever duplication of memories [17], trie approaches by splitting the rulesets [13].

B. Aggregation

To increase the throughput of the aggregation we want to minimize the sizes of the crossproducts that DCFL produces. We want to remove rules that contribute the most to the crossproducts. Those are usually the most general rules, as those match every packet and need to be accounted for. We can determine these most interfering rules by analyzing the rulesets. Once we find which rules are the most interfering we can offload those rules into a small TCAM that runs in parallel with the DCFL.

The sizes of crosspruducts are given by the sizes of the partial matches R^{i+1} and $R^{1,\dots,i}$. To optimize the approach, we want to determine which rules should be removed to decrease these values. First, we build tries T^i that represent the rules in the ruleset for each dimension *i*. Each trie is essentially a binary tree where paths via edges represent prefixes that are being matched in the given dimension (each edge has a value of 0 or 1 assigned to it). Afterwards, we can also build tries $T^{i,j,\cdots}$ for multiple dimensions. These are constructed by building a trie for the dimension *i* and for each rule from this trie adding an empty transition to a trie built for dimension *j*, but only from rules in dimension *j* that are

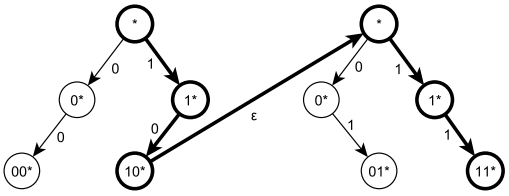

Fig. 2. Multidimensional trie for ruleset analysis.

ever combined with the original rule from the dimension *i*. Figure 2 illustrates this. For example, the highlighted nodes and edges represent the rule *(10*,11*)*.

Based on these tries we can compute some inference value $I^i(r)$ for every rule *r*. This inference value represents how many other rules can be matched along with this rule in the given dimension *i* and as such represents the size of possible partial result R^i that this rule can be involved in. The value can be computed by analyzing the tree as it corresponds to the maximal number of rules encountered on any path that starts from the root and goes through the node that represents the rule *r*. In the same fashion, we can also define and compute $I^{i,j,\cdots}(r)$ from $T^{i,j,\cdots}$. Value $I^{i,j,\cdots}(r)$ gives us the possible maximal size of the set of results that can match a packet in dimensions *i,j,....* Additionally, we can define interference with descendants only as $I^i_{des}(r)$ and $I^{i,j,\cdots}_{des}(r)$. The value corresponds to the maximal number of (more specific) rules encountered after encountering the node representing rule *r* on any path that starts from the root and goes through the node that represents the rule *r*.

Building and analyzing multidimensional tries becomes exponentially harder with the number of dimensions used. If we want to traverse any sub-tree of a node in first dimension *i* it can potentially have $|R| * w_i$ nodes, where $|R|$ is the ruleset size and w_i is the bit width of that dimension. Each of these nodes can then be linked with a tree in the second dimension *j* that might again have $|R| * w_j$ nodes that are all linked with a tree in another dimension and so on. Building the whole trie from *R* rules can then take up to $O(|R|^{d+1} * w_m^d)$.

To analyze a ruleset using the built tries we use algorithm 1. We start by building one-dimensional tries T^i for each dimension. Then we also build multi-dimensional tries $T^{1,\dots,i}$.

Now we can find the maximal sizes of sets of partial matches that can enter each aggregation unit (V^i represents all nodes of T^i), and based on those also find the maximal size of crossproduct (lookups of the aggregation unit) that DCFL needs to check and a dimension in which this happens (prime candidate for optimization):

$$\begin{aligned} R^i_{max} &= max_{v \in V^i}(I^i(v)) \\ R^{1,\dots,i}_{max} &= max_{v \in V^{1,\dots,i}}(I^{1,\dots,i}(v)) \\ c &= max_{i=1,\dots,D-1}(R^{1,\dots,i}_{max} \times R^{i+1}_{max}) \\ i_o &= argmax_{i=1,\dots,D-1}(R^{1,\dots,i}_{max} \times R^{i+1}_{max}) \end{aligned} \tag{1}$$

If we want to decrease c we want to decrease either $R^{1,\dots,i_o}_{max}$ or $R^{i_o+1}_{max}$. To decrease the value of $R^{i_o+1}_{max}$ we sort

Algorithm 1 Algorithm for analyzing rulesets using D dimensions.

while TCAM not full **do**
  for $i = 1$ to D **do**
    Build T^i
    Compute I^i and I^i_{des}
    Find max. size of partial match R^i_{max}
  end for
  for $i = 2$ to $D - 1$ **do**
    Build $T^{1,\dots,i}$
    Compute $I^{1,\dots,i}$ and $I^{1,\dots,i}_{des}$
    Find max. size of partial match $R^{1,\dots,i}_{max}$
  end for
  Find i_o with max. value of $R^{1,\dots,i_o}_{max} \times R^{i_o+1}_{max}$
  if Better to decrease $R^{i_o+1}_{max}$ **then**
    Sort nodes of T^{i_o+1} based on $(I^{i_o+1}, I^{i_o+1}_{des})$
    Offload all of the rules corresponding to nodes with highest $(I^{i_o+1}, I^{i_o+1}_{des})$
  else
    Sort nodes of $T^{1,\dots,i_o}$ based on $(I^{1,\dots,i_o}, I^{1,\dots,i_o}_{des})$
    Offload all of the rules corresponding to nodes with highest $(I^{1,\dots,i_o}, I^{1,\dots,i_o}_{des})$
  end if
end while

all of the nodes of trie T^{i_o+1} in descending order based on primarily values of I^{i_o} and secondarily on values of $I^{i_o}_{des}$. Then we can find all of the nodes, or rather their corresponding rules, that have the highest value of I^{i_o}. Each of those nodes is on some critical path. Note that there can be multiple different critical paths within the trie. If we want to decrease $R^{i_o+1}_{max}$ we have to remove at least one rule from every critical path. To determine if two nodes are part of the same critical path we can simply look at the value of $I^{i_o}_{des}$ - if the value is the same for both then they have to be part of different critical path. If they were part of the same path then one would have to be a prefix (generalization) of the other and as such the other would show up in its $I^{i_o}_{des}$ and this value would be higher by at least one. Now we just remove all of the rules corresponding to nodes with the highest value of I^{i_o} that also have the highest value of $I^{i_o}_{des}$. By doing this we lowered the length of each critical path (and also $R^{i_o+1}_{max}$) by 1. Note that each node in the trie might correspond to multiple rules Decreasing the value of $R^{1,\dots,i_o}_{max}$ works the same, but we use $T^{1,\dots,i_o}$, $I^{1,\dots,i_o}$ and $I^{1,\dots,i_o}_{des}$.

After offloading these rules we can recompute all of the tries and interferences and repeat the process to decrease c further. Trying to offload rules for the most general node (one with the highest I_{des}) might not always be the best solution. The length of the critical path might be decreased by offloading rules corresponding to any other node along this path. However, in general, it can be assumed that removing the most general rules can lead to better results when doing the following iteration.

Since the entire process of computing full interference for every dimension and possible multidimensional combination needs to be done multiple times, the entire analysis becomes even more complex and slow. To decrease the computational complexity one of the following can be used - analyzing only single-dimensional tries in $O(d * |R| * w_m)$ (since we don't build any multidimensional tries) or using generality (for example, number of masked bits) of the rule instead of interference in $O(|R|)$.

Finally, one additional optimization could be to account for the frequency of each rule being hit and optimize mainly the rules or rather paths that are hit the most. This introduces even more complexity and a requirement to keep stateful information about rules being hit.

Name	Dimensions	Number of rules	Overlap
acl1	4	2406	high
fw1_05_05	5	733	medium
fw2_05_05	4	941	low

TABLE I
CHARACTERISTICS OF RULESETS USED.

IV. RESULTS

To analyze the effectiveness of our optimizations we used 3 main sets of classification rules. These sets were generated by ClassBench [20] tool and are part of NetBench [21] framework. Table I shows the characteristics of the sets, main one being level of overlap between ranges or prefixes within rules. *acl1* is an example of ruleset with many overlapping ranges of port values, therefore many different rules can match the same packet within single dimension, which leads to DCFL not being very effective. *fw2_05_05* on the other hand has little overlap between rules and prefixes thus DCFL shows much better results. Additionally, *fw1_05_05* represents a middle ground.

Computation of the full rule analysis might not be feasible on the fly as previously mentioned in section III-B. Therefore we consider 3 different approaches to rule offloading. For each approach a TCAM with 32 entries is used.

First approach is to ignore interference altogether and simply approximate it by generality (the length of prefix or number of masked bits) of the rule. The approach works on the fly and waits until a certain number of rules were added and then offloads rules that are more general than the ones seen before. This is essentially a secretary problem, therefore the best results are achieved when the number of initial rules is $|R|/e$ [22]. To make results even better, we can ignore and immediately offload some of the outliers (rules that are too general) as things like default rules are usually added first. The second approach works on-the-fly in the same way, but uses the interference described in section III-B instead of simple generality. Finally, the last approach knows the entire ruleset in advance and does full interference analysis on the full ruleset.

Figure 3 shows the results achieved by the different approaches in the worst-case scenario. For every approach, we

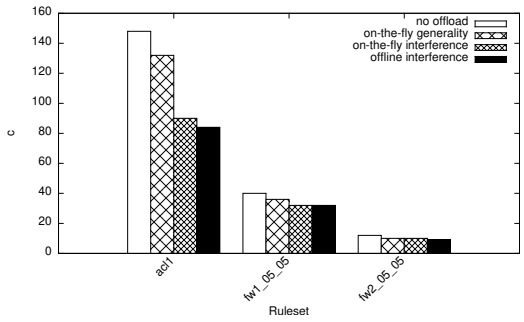

Fig. 3. Optimizations achieved for DCFL for worst-case.

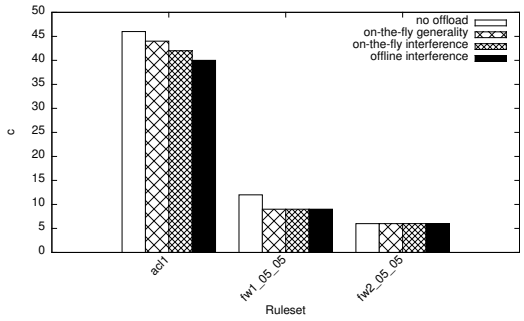

Fig. 4. Optimizations achieved for DCFL for mean-case.

can see the worst value of c which represents the maximal crossproduct size and therefore basically the number of clock cycles needed to classify the packet. To illustrate throughput increase even better, figure 5 shows the percentage increase in the throughput of an optimized DCFL architecture. We can see that the full analysis fares the best with interference analysis being close behind. However, even simple generality-based offloading provides considerable results. For less complex rulesets with smaller overlap (*fw1_05_05* and *fw2_05_05*) the decrease is around 20 % which corresponds to an increase in throughput of around 25 %. As expected the highest benefit is seen when a more complex ruleset is analyzed (*acl*). In this case, a decrease of 43 % can be observed which leads to an increase in throughput of 76 %.

A similar situation can be seen for the mean-case scenario shown in figure 4. Full analysis still outperforms the other approaches. In this case the benefits are not as high, since removing the most interfering rules breaks the critical paths which are not necessarily hit by every single packet. However, the speedup can still be up to 33 %.

The increase in throughput can be converted into a decrease in needed resources. Any aggregation node of the DCFL architecture can be duplicated to increase throughput. Combining the optimizations presented in this paper with aggregation node duplication can lead to fewer units being duplicated. Table II shows the crossproduct size for each aggregation unit in the worst-case scenario for ruleset *acl*. Assuming that we want to process one packet every 50 clock cycles we would need a total of 8 aggregation units if no optimization was used.

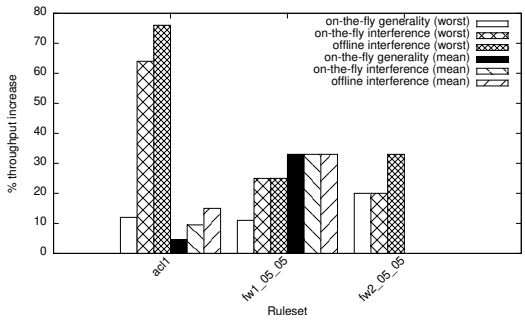

Fig. 5. Throughput increase achieved for different configurations.

Optimization	c^1_{max}	c^2_{max}	c^3_{max}
no offload	54	148	148
on-the-fly generality	49	132	102
on-the-fly interference	46	90	90
offline interference	45	84	81

TABLE II
WORST-CASE CROSSPRODUCT SIZES FOR AGGREGATION UNITS WHEN RULESET *acl* IS USED.

Using even on-the-fly interference approach only requires 5 total aggregation units We can therefore use the 64 % increase in throughput to achieve 37 % decrease in resources needed to carry out the aggregation.

Figures 6 and 7 show scaling of block RAMs and logic needed for increasing throughput of the optimized architecture in the mean-case for rulesets *fw1_05_05* and *acl* compared to other approaches. The results were obtained on Kintex-7 family chip and the architecture runs at 200 MHz and throughput is converted from cycles needed to gigabits per second for the shortest possible packets (64B). Note that the architecture has capacity of around 5500 rules, whereas other architectures may have different rule capacities (mainly decision forest has capacity of 10 000 rules). Additionally, our architecture used quite small TCAMs (since number of unique values in single dimensions is a lot lower than number of rules) as engines for searching in some dimensions. We can see that the resource requirements stay manageable even for high throughputs. The optimized approach is comparable to or even better than other approaches in terms of block RAMs needed, while being also comparable in logic required. The main benefit is good scalability and flexibility. Optimized DCFL can be efficiently used for different use cases and be scaled based on the required throughput and available resources to fit the specific use case as best as possible, leaving more FPGA resources for other network functionality.

V. CONCLUSION

The paper presents several optimizations for packet classification on FPGAs. We have shown that the proposed optimizations increase throughput and decrease memory and logic requirements. Due to the flexibility of optimized DCFL architecture, we can provide more efficient mapping of multiple

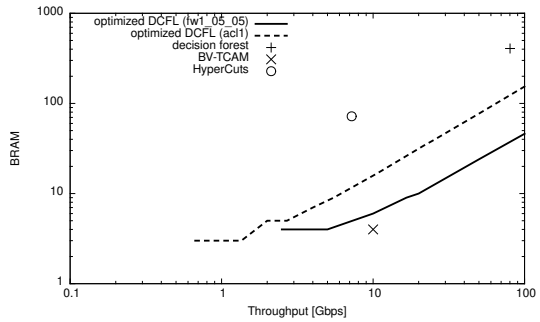

Fig. 6. BRAMs used by optimized DCFL compared to other approaches for mean-case.

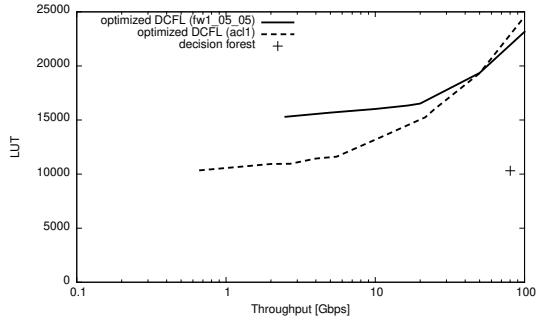

Fig. 7. Scaling of LUTs used by optimized DCFL for mean-case.

Match/Action tables and other packet processing from P4 language to FPGA without affecting throughput or rule capacity.

The proposed optimizations can increase packet classification throughput by up to 76 % compared to the standard 5-tuple based DCFL algorithm. When combined with memory duplication, we can also decrease memory requirements. For the same throughput and table capacity, the proposed optimization decreases the hardware resources by up to 37 %. All these improvements are at the cost of more complex rule adding and one additional small TCAM memory. Nevertheless, the benefits significantly outweigh these downsides. They provide an efficient way to tailor packet classification for specific applications and make this task less resource-intensive. More chip area remains for other packet processing or network functionality.

ACKNOWLEDGMENTS

This work was supported by Brno University of Technology under project number FIT-S-23-8141.

REFERENCES

[1] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[3] P. Benáček, V. Puš, and H. Kubátová, "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.

[4] P. Benáček, V. Puš, H. Kubátová, and T. Čejka, "P4-to-VHDL: Automatic generation of high-speed input and output network blocks," *Microprocessors and Microsystems*, vol. 56, pp. 22 – 33, 2018.

[5] D. Taylor and J. Turner, "Scalable packet classification using distributed crossproducing of field labels," in *24th Annual Joint Conference of the IEEE Computer and Communications Societies*, 2005, pp. 269–280.

[6] H. Mahmood, Z. Ullah, O. Mujahid, I. Ullah, and A. Hafeez, "Beyond the limits of typical strategies: Resources efficient fpga-based tcam," *IEEE Embedded Systems Letters*, vol. 11, no. 3, pp. 89–92, 2019.

[7] F. Syed, Z. Ullah, and M. K. Jaiswal, "Fast content updating algorithm for an sram-based tcam on fpga," *IEEE Embedded Systems Letters*, vol. 10, no. 3, pp. 73–76, 2018.

[8] I. Ullah, Z. Ullah, U. Afzaal, and J.-A. Lee, "Dure: An energy- and resource-efficient tcam architecture for fpgas with dynamic updates," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 6, pp. 1298–1307, 2019.

[9] 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, 2005, pp. 238–245.

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

[11] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2003, pp. 213–224.

[12] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 191–202, 1998.

[13] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, 2012.

[14] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms - ESA 2001*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, vol. 2161, pp. 121–133.

[15] A. Kirsch, M. Mitzenmacher, Y. Baohua, X. Yibo, and L. Jun, "Using a queue to de-amortize cuckoo hashing in hardware," 2007. [Online]. Available: http://www.eecs.harvard.edu/~michaelm/postscripts/aller2007.pdf

[16] L. Kekely, M. Žádník, J. Matoušek, and J. Kořenek, "Fast lookup for dynamic packet filtering in FPGA," in *17th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. Warszaw, Poland: IEEE Computer Society, 2014, pp. 219–222, iSBN: 978-1-4799-4558-0.

[17] M. Kekely, L. Kekely, and J. Kořenek, "General memory efficient packet matching fpga architecture for future high-speed networks," *Microprocessors and Microsystems*, vol. 73, no. 3, pp. 1–12, 2020.

[18] 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*. New York, NY, USA: ACM, 2006, pp. 61–70.

[19] M. Kekely and J. Korenek, "Packet classification with limited memory resources," in *2017 Euromicro Conference on Digital System Design*. Institute of Electrical and Electronics Engineers, 2017, pp. 179–183.

[20] D. Taylor and J. Turner, "Classbench: a packet classification benchmark," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 3, 2005, pp. 2068–2079 vol. 3.

[21] V. Pus, J. Tobola, V. Kosar, J. Kastil, and J. Korenek, "Netbench: Framework for evaluation of packet processing algorithms," in *2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, 2011, pp. 95–96.

[22] E. B. Dynkin, "Optimal choice of the stopping moment of a markov process," *Dokl. Akad. Nauk SSSR*, vol. 150, no. 2, pp. 238–240, 1963.