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/© 2019 Published by Elsevier B.V.

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

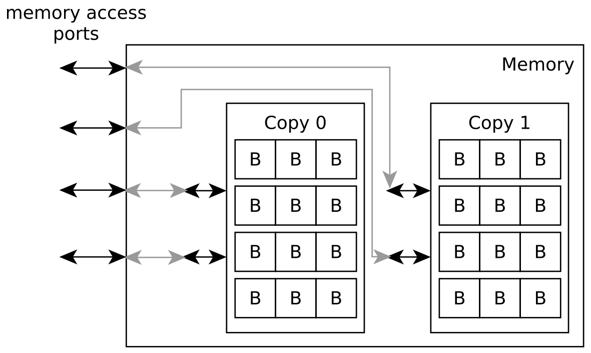

Fig. 1. The memory architecture of simple replication approach using memory tiles on FPGA.

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)

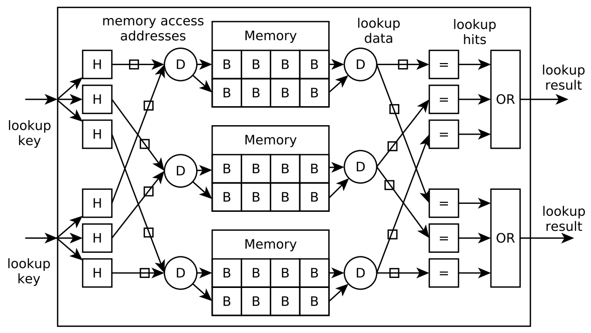

Fig. 2. The top-level architecture of the proposed optimization approach.

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:

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

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:

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

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:

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

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:

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

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

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

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:

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

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:

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

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

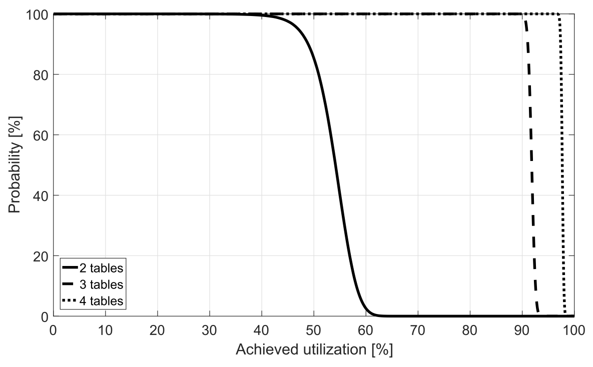

Fig. 3. Achieved memory capacity utilization for a different number of hash tables (functions).

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

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

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.

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 × 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 × 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 × 1024 items) while Xilinx BlockRAM tile has 36 Kb (36 b × 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

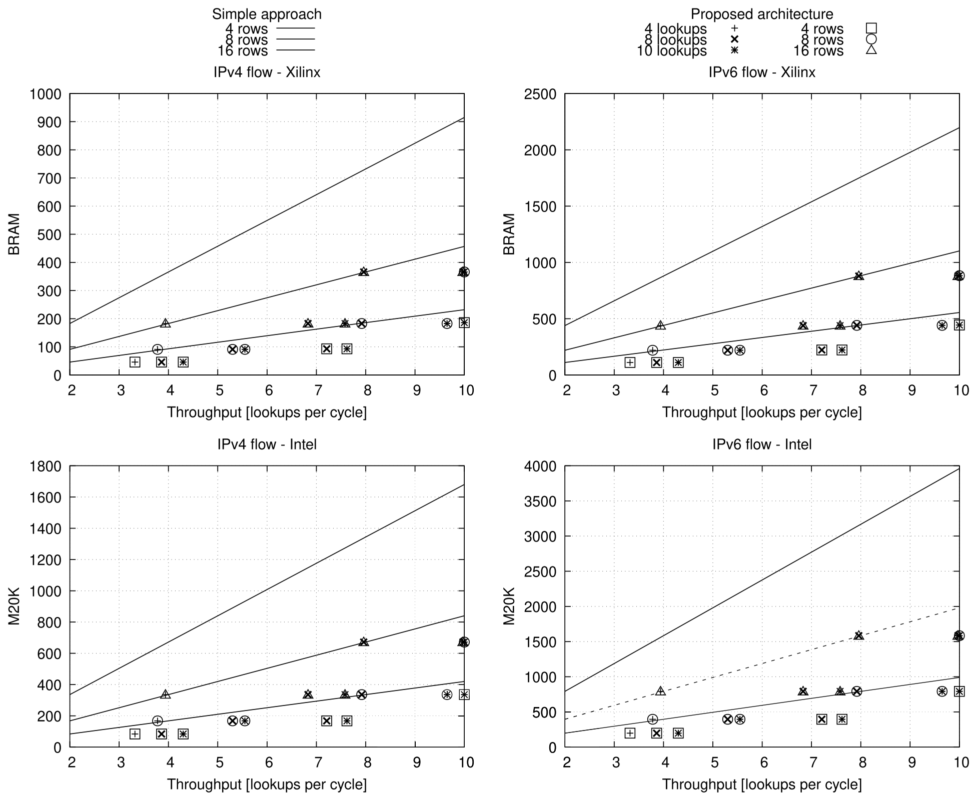

Fig. 4. The relations between utilized memory tiles and achieved throughput for different FPGAs and key widths when using 3 hash functions.

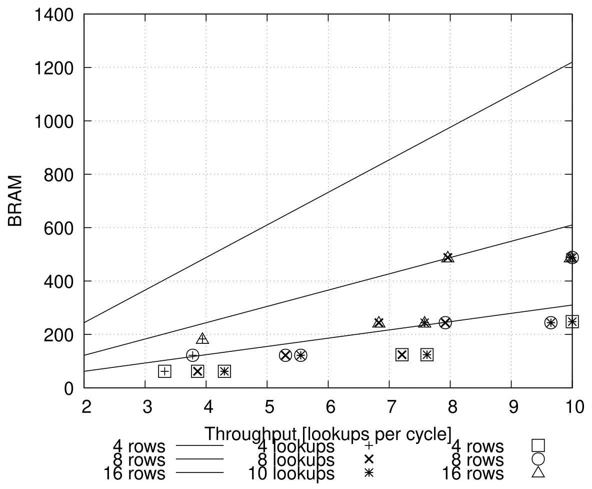

Fig. 5. The relation between utilized memory and achieved throughput for Xilinx FPGAs and IPv4 flows when using 4 hash functions.

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

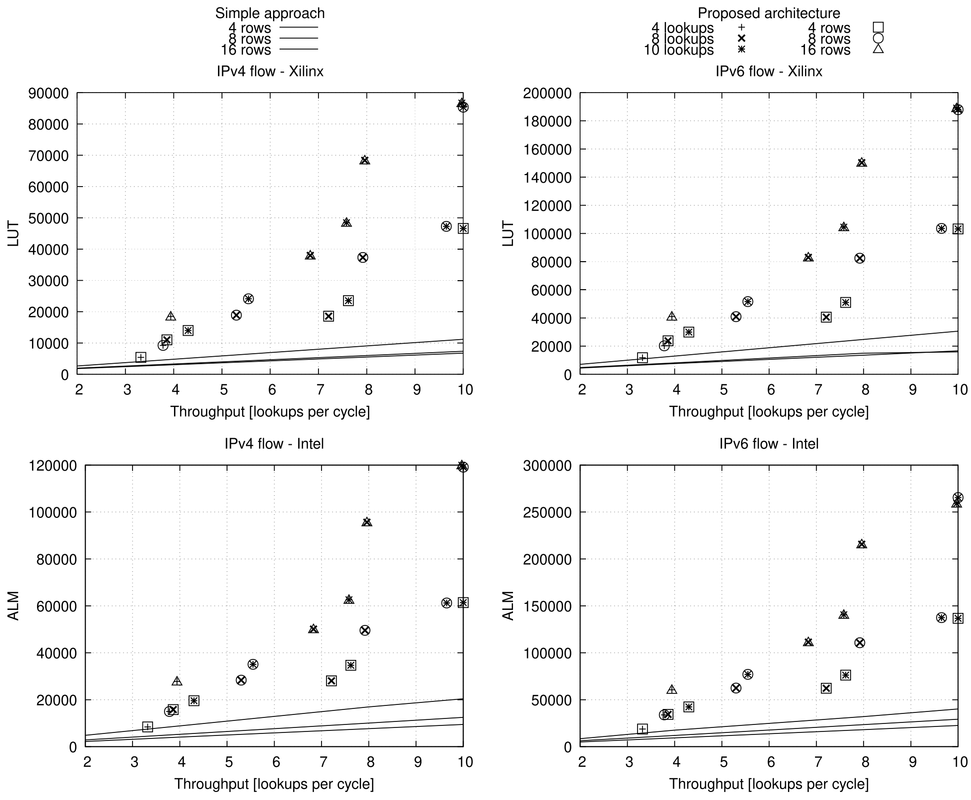

Fig. 6. The relation between utilized logic and achieved throughput for different FPGAs and key widths when using 3 hash functions.

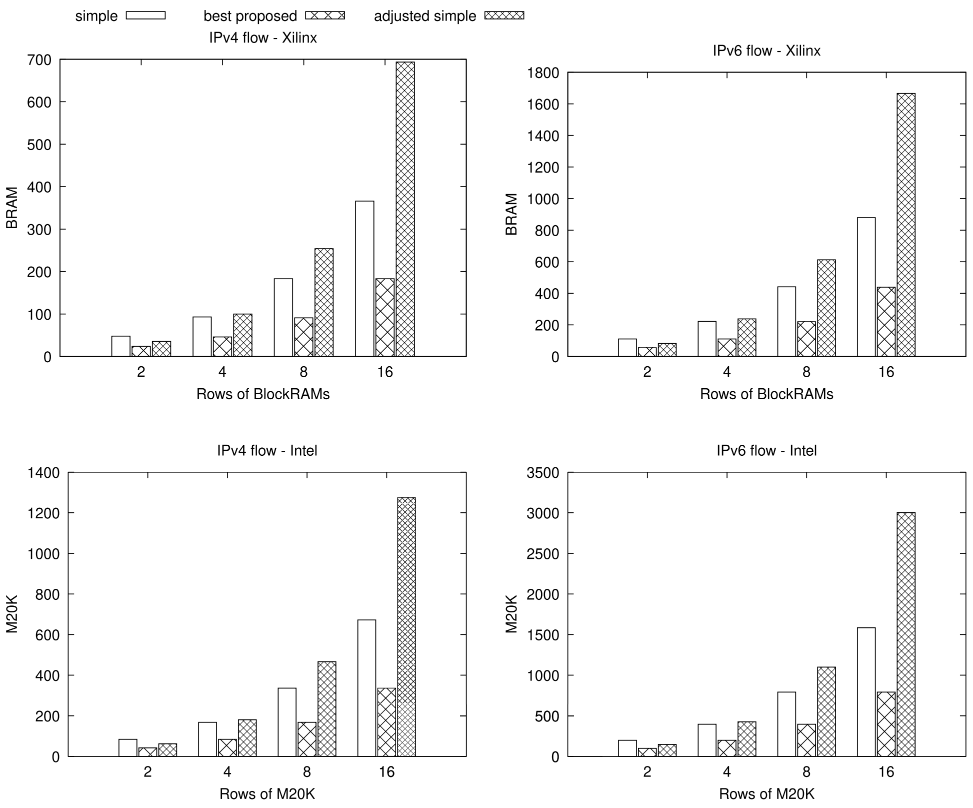

Fig. 7. Memory requirements comparison when achieving at least 800 Gbps.

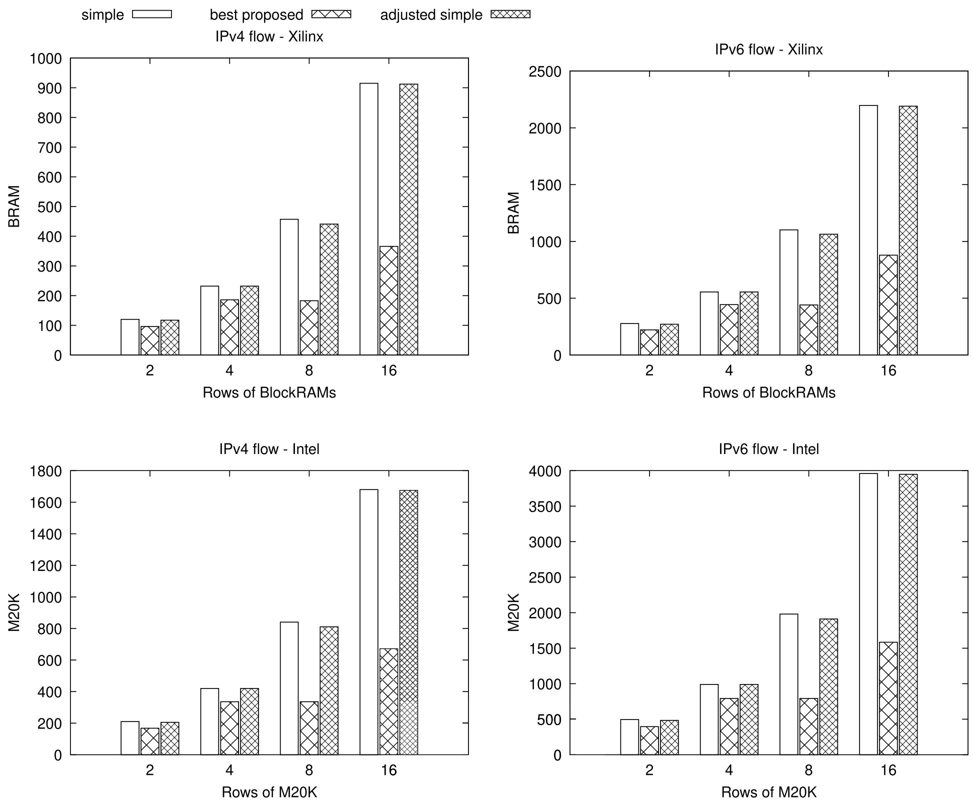

Fig. 8. Memory requirements comparison when achieving at least 2.4 Tbps.

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.

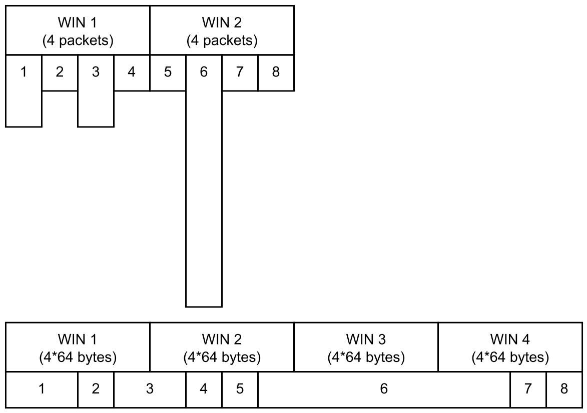

Fig. 9. Illustrations of examined scenarios with worst-case packet window and average-case byte window.

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

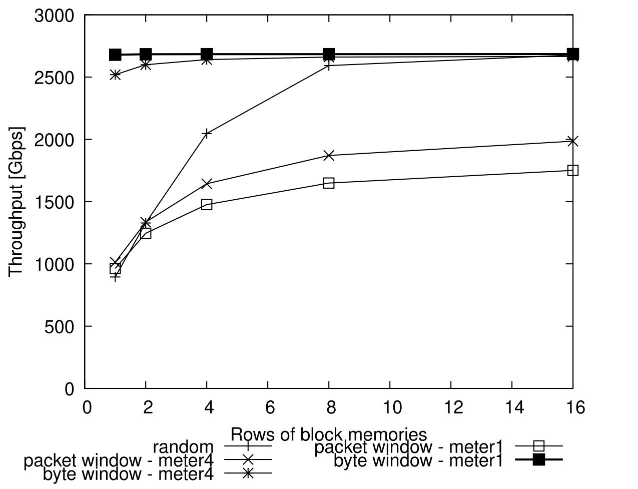

Fig. 10. Throughput results on real network traces for architecture with 10 lookups and 2 × memory replication.

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.

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

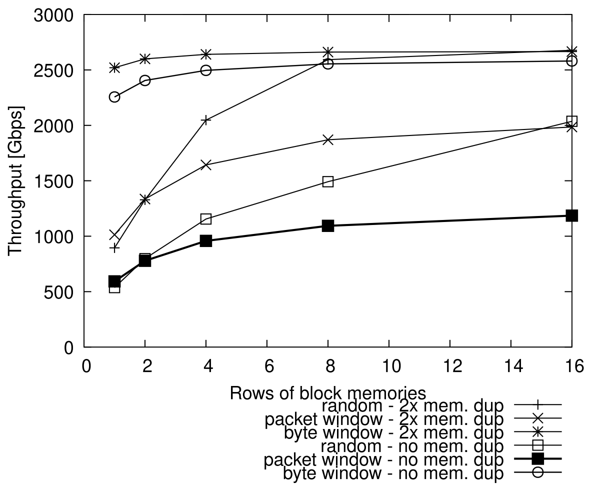

Fig. 11. Throughput results on real network trace *meter4* for architecture with 10 lookups and different levels of memory replication.

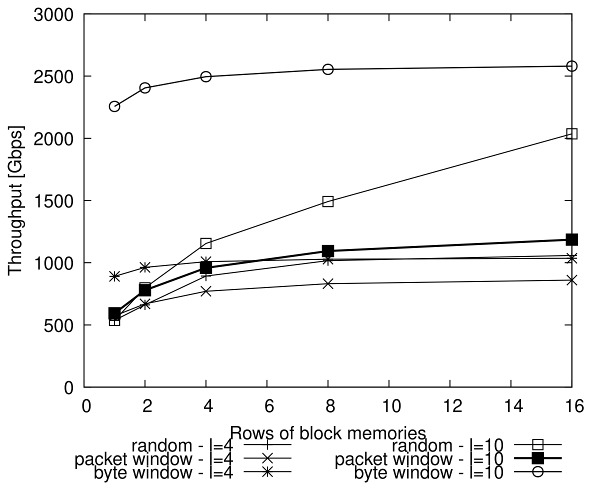

Fig. 12. Throughput results on real network trace *meter4* for architecture with different numbers of lookups.

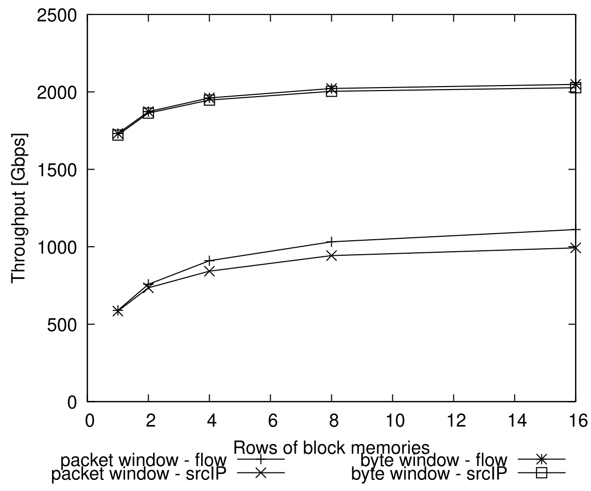

Fig. 13. Throughput results for architecture with 8 lookups and no memory replication with different keys.

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 deamortize 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.