Efficient Packet Classification on FPGAs also Targeting at Manageable Memory Consumption

Nitesh Guinde, Sotirios G. Ziavras and Roberto Rojas-Cessa
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102, USA

ABSTRACT
Packet classification involving multiple fields is used in the area of network intrusion detection, as well as to provide quality of service and value-added network services. With the ever-increasing growth of the Internet and packet transfer rates, the number of rules needed to be handled simultaneously in support of these services has also increased. Field-Programmable Gate Arrays (FPGAs) provide good platforms for hardware-software co-designs that can yield high processing efficiency for highly complex applications. However, since FPGAs contain rather limited user-programmable resources, it becomes necessary for any FPGA-based packet classification algorithm to compress the rules as much as possible in order to achieve a widely acceptable on-chip solution in terms of performance and resource consumption; otherwise, a much slower external memory will become compulsory. We present a novel FPGA-oriented method for packet classification that can deal with rules involving multiple fields. This method first groups the rules based on their number of important fields, then attempts to match two fields at a time and finally combines the constituent results to identify longer matches. Our design with a single FPGA yields a larger than 9.68 Gbps throughput and has a small memory consumption of around 256Kbytes for more than 10,000 rules. This memory consumption is the lowest among all previously proposed FPGA-based designs for packet classification. Our scalable design can easily be extended to achieve a higher than 10Gbps throughput by employing multiple FPGAs running in parallel.

Keywords
FPGA, Packet Classification, Pattern Matching.

1. INTRODUCTION
In general, Internet routers targeting at best-effort services forward packets based on a first-come first-served basis, where the same quality of service is provided to all packets. However, recent developments require routers to provide various quality-of-service levels and also support various functions, such as admission control, resource reservation, per-flow queuing, and fair scheduling. Additional services, among others, include Virtual Private Network (VPN), distributed firewalls, IP security gateways and traffic-based billing. These enhancements require packet classification involving the comparison of header fields in incoming packets against a known set of rules; such fields are the source IP, destination IP, source port, destination port and protocol. Upon a successful match, the system performs the predefined action on the incoming packet as suggested by the rule. In packet classification, three types of matching can be performed with these fields: 1) Exact matching: the fields in the header of the packet should match exactly the respective fields in the rule. 2) Prefix matching: a prefix and a prefix mask are provided by the rule for the longest match with fields in the packet. 3) Range matching: A range is provided by the rule and a match is found if the header information in the packet falls within this range.

Due to the rapid growth in the size of rule sets for packet classification and increases in the link rate, multi-field based packet classification has become a fundamental challenge for high-speed designs. Of course, a software-oriented solution cannot satisfy the required processing speeds. Hence, efficient hardware implementations using ASIC devices or FPGAs [22] have recently received substantial attention [2-5]. Some of the best packet classification algorithms targeting at the matching of d fields, where $d>3$, have $O(log\ n)$ time complexity at the cost of $O(n^d)$ space, or $O((logn)^{d-1})$ search time at the cost of $O(n)$ space, where n is the number of stored rules [16].

Researchers usually apply either an algorithmic or an architectural solution towards high performance in packet classification. However, sometimes hybrid approaches are pursued that involve a combination of these solutions. Such solutions are typically classified as *decomposition based* [12, 13] or *decision-tree based* [1, 7]. The former solutions search multiple fields in parallel and then combine the results, thus they are good candidates for hardware implementation. Decision-tree based solutions, like Hi-cuts [1], take a geometric view of the packet classification problem. The search space is reduced at each node of the decision tree based on information from one or more fields in the rule. A decision tree is built by choosing a dimension (i.e., a field) and also the number of cuts to make in the chosen dimension by using local optimization decisions. At every node a cut involves one field. The cutting process is performed at each level and recursively on the children at the next level until the number of rules associated with each node at a level falls below a threshold.

FPGAs are bound by their limited number of resources, a disadvantage which is exacerbated further by continuous increases in rule-set sizes that beg for scalable hardware solutions. This implies the need for high on-chip rule-set compression that could ideally eliminate all off-chip memory accesses. Although packet classification has been recently researched by various groups, our ultimate goal is to achieve a comprehensive on-chip FPGA-based solution that does not require external memory accesses, while also being able to efficiently support more than 10,000 rules with a currently available FPGA device.

We first split the rule-set into groups depending on their number of valid fields; a field is valid for a given rule if its value must be matched for this rule to be activated. We then break up at static time each rule in a group into patterns involving two fields; if a rule contains just one valid field, it is split into smaller

patterns. The run-time system attempts to match pairs of fields at a time and these matching results are then combined to produce an address in an on-chip memory where a complete match, if present, can be verified. Using position-based vectors and partial matches, the groups are searched in parallel for the incoming packet headers and a running sum is generated that depends on the position vectors. This sum generates a unique address in the memory that verifies a match if this sum matches a pre-stored value. A final decision also involves rule priority. Compared to existing hardware-oriented solutions, our solution reduces drastically the on-chip memory consumption while also being able to match network speeds. It is also scalable, allowing the incorporation of many fields in the packet classification process.

2. RELATED WORK

A scheme that searches in parallel every individual field of every rule using a prefix trie was proposed in [13]; the result is a bit vector (BV) with each bit representing a rule. A bit is set if the corresponding rule is matched in this field; it is reset otherwise. The intersection of all BVs via their bit-wise AND operation indicates the rule that matches the incoming packet. Although the scheme provides high throughput, the memory efficiency is very low, which makes it cumbersome for large rule sets. The work in [12] employed the BV scheme assuming that for large rule-sets a packet can match not more than a few rules; a new aggregated bit vector scheme was devised using recursive aggregation of bit maps, and rule rearrangements for less memory space and higher speed. Ternary content addressable memories (TCAMs) [14, 15] also have been used in classification. Nevertheless, TCAMs are not easily scalable in terms of clock rate, power consumption or circuit area. Most of the TCAM-based solutions also have difficulty when ranges in fields have to be converted into prefixes. By combining TCAMs and the BV algorithm, Song et al. [10] presented the BV-TCAM architecture for packet classification. A TCAM is used for prefix or exact matches, whereas a multi-bit trie implemented as a Tree Bitmap [11] is used for source or destination port lookup. Their analysis involved 222 SNORT header rules.

Bloom filters [5, 8, 9] are popular due to their constant time requirements and low memory consumption. However, false positives are possible that require a secondary off-chip memory to check the authenticity of potential matches at a much slower rate. Trie-based schemes, like hierarchical tries, set-pruning tries [19] and grid-of-tries [20] work well for two-dimensional classifiers; however, as the number of dimensions increases their complexity increases too. The works in [17-18] surveyed a large number of classification techniques. They concluded that very rarely will a packet match multiple rules. Using heuristics with real databases, [17] developed the Recursive Flow Classification (RFC) algorithm that splits the packet header into many chunks of contiguous bits and then represents each one of them with a reduced number of action bits. A recursive process of action combining at run time yields the final action to be performed on the packet. The amount of storage for RFC increases rapidly as the classifier size increases; it uses about 4 Mbytes to store 15,000 rules. Also, the heuristics-based HiCuts [1] approach has low memory requirements, consuming around 1 Mbyte for 1700 rules.

Distributed Crossproducting of Field Labels (DCFL) in [6] uses a decomposition-based algorithm that employs independent search engines for different fields. They perform a distributed set membership query using a network of aggregation nodes; each query performs an intersection on the set of possible field combinations matched by the packet and the set of field combinations specified by filters in the filter set. Their design makes use of Bloom filters, thus it suffers from false positives. A memory-efficient decomposition-based packet classification algorithm in [13] employs multi-level Bloom filters to combine the search results from all the fields. Their FPGA implementation, called 2sBFCE [5], can support 4K rules with 178 Kbytes of memory consumption. It takes an average of 26 clock cycles per packet, resulting in a low throughput of 1.875 Gbps.

Hypercuts [7] is another heuristics-based approach similar to Hi-cuts except that it allows cuttings on multiple fields per step, thus resulting in a fatter and shorter decision tree. The hypercuts algorithm was modified in [2] to build a pipelined balanced-tree design. They reduce rule duplications in hypercuts by using an internal node to store the rules which are present in all of its children nodes and by also cutting precisely the range of to-be-matched fields. The design involves a tree pipeline that forms a decision tree and a rule pipeline that contains the rule lists for a node. Their implementation of 10K rules consists of 11 tree pipeline stages, 8 rule pipeline stages and a total of 12 rule pipelines. The memory consumption of their design is quite substantial, using around 407 Xilinx Virtex-5 block RAM (BRAM) memories (36Kbits per BRAM) out of which 612 Kbytes are taken by the rule lists and the pipelined tree configuration.

3. OUR METHOD

We treat the packet classification problem as a pattern matching problem involving two fields of the packet header at a time. Let us assume four consecutive fields F1, F2, F3 and F4 in a rule. Our method compresses at static time the information that appears in every possible pair of fields in the rule-set by extracting fragments of 8 and 7 contiguous bits, and then encoding the position of prestored 8-bit and 7-bit patterns in these pairs using position/bit vectors. An accumulated sum also is created for individual fields by adding up unique weight tuples previously assigned to 8-bit and smaller fragments in the fields of the rule-set. For example, for a source IP field with 32 bits in a rule we create 32 distinct accumulated sums. Using these position vectors, we can identify multiple matches of field pairs (i.e., F1-F2, F1-F3, F1-F4, F2-F3, F2-F4 or F3-F4) simultaneously.

We can then combine individual match results for field pairs by adding up individual field sums to produce a combined sum; the ultimate verification for a rule match is successful if the combined summation value is the same as the pre-stored value in the FPGA's BRAMs for this rule. The BRAM address for the rule is a function of the combined sum when considering all of its valid fields. Let us now look at a brief example of packet classification using our method. Assume the packet classification rules of Table 1 that involve five fields: source IP, destination IP, source port, destination port and protocol. The number following the '/' in the SIP and DIP fields is the mask which signifies the number of rule-valid bits in the field. Since the protocol normally requires exact matching, we do not employ it in our pre-processing for the creation of position vectors.

.
.

Table 1. Sample Rule Set

ID	Source IP (SIP)	Destination IP (DIP)	Source Port (SP)	Destination Port (DP)	Protocol 8 bits	Group 4 bits
1	90.24.13.4/32	51.63.17.19/32	1023 :1024	413 :413	TCP	15
2	89.24.13.4/32	5.6.7.9/32	1023 :1024	103 :109	TCP	15
3	11.71.19.14/23	23.98.128.80/21	0 :1023	0 :5	TCP	15
4	163.92.37.190/32	11.24.179.56/32	0 : 65535	8080:8080	TCP	13
5	97.166.41.112/31	182.125.194.192/23	0 : 65535	2930 : 2930	TCP	13
6	27.26.30.130/19	246.67.55.211/25	0 : 65535	3106 : 3108	TCP	13
7	19.14.103.41/32	1.6.0.0/0	1023 :1024	100 :101	TCP	11
8	1.6.0.0/0	5.6.7.9/32	1023 :1024	13 :13	TCP	7
9	101.121.77.33/25	23.98.128.80/21	0 : 65535	0 :65535	TCP	12
10	11.23.131.145/29	5.6.7.9/0	0 :1023	0 :65535	UDP	11
11	0.0.0.0/0	0.0.0.0/0	0:1023	13:13	TCP	3
12	0.0.0.0/0	0.0.0.0/0	0 :65535	0 :65535	Any	0

3.1 Pre-processing Phase

Rule grouping: We first separate the rules into groups during pre-processing based on their number of valid fields. For example, the first rule in Table I has all of the four fields valid and, hence, is placed in group G15 (i.e., "1111" in binary is "15" in decimal). Similarly rule 4 has the first, second and fourth fields valid and, hence, is placed in group "1101" or G13.

Fragmentation schemes: The binary representation of the value in each field of each rule is then fragmented using two schemes that involve eight and seven contiguous bits, respectively. We denote these fragmentation schemes as FRAG8 and FRAG7, respectively. Using two fragmentation schemes aids the process of filtering out falsely generated bit vector matches. We explain this process in detail later on in this Section.

FRAG8 and FRAG7 split the field pattern into 8-bit and 7-bit fragments, respectively, till the tail is reached. Fig. 1 shows the first field of rule 3 being fragmented according to both schemes. Since this field in rule 3 contains 23 valid bits, the tail will contain 7 bits. This fragmentation is then used to create a Bit Vector (BV) and an End Vector (EV) for every fragment value/pattern obtained. More specifically, BV shows the position of this fragment pattern in the field, actually for all the rules that eventually contain it, excluding their tail. That is, if a particular fragment pattern appears only in positions 1 and 3 of the same field in the same or different rules, then its BV will be "0110...0".

The EV vectors store information about the tail fragments of patterns in fields. If a fragment appears as a tail, then it will contain '1' in the respective position of its EV vector. Multiple appearances of a fragment pattern in the same position (tail or non-tail) of multiple rules are registered only once in this pattern's BV or EV. The lengths of BV and EV depend on the fragmentation scheme, and also on the length of the field in bits. In Fig. 1, the field shown is the Source IP which is a 4-byte field and hence the BV generated for it will be 3 bits long under FRAG8 and 4 bits long under FRAG7 (since tail fragments are not included in BV). Similarly, the length of EV will be 4 bits and 5 bits under FRAG8 and FRAG7, respectively. In Fig. 1, the decimal value/pattern 11 appears in the first fragment and hence its BV will be "100" and the pattern 23 has a BV of "010" since it appears in the second 8-bit position (with just one rule in the set).

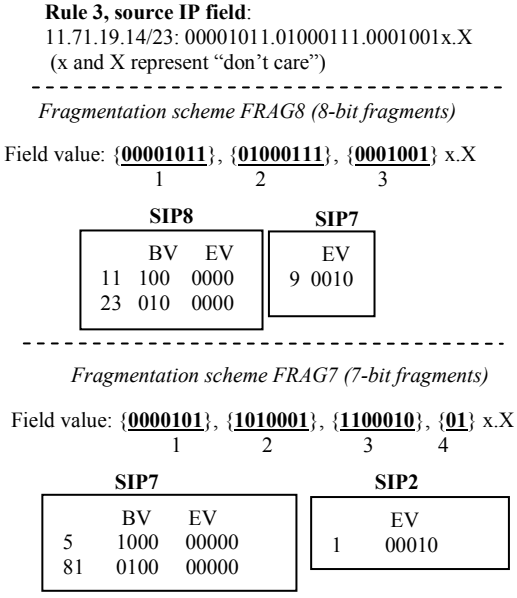

Figure 1. Applying FRAG8 and FRAG7.

For a given fragmentation scheme, say FRAG8, we store these BV and EV vectors in separate tables corresponding to their lengths. The BV and EV vectors of the SIP field for fragments of 8 bits are stored in table SIP8. Similarly, we store the EV of tail fragments having i bits in table SIPi, where i=1,2,…,7. SIP7-SIP1 only contain EVs since these fragments will appear only as tails. This same procedure is carried out for the FRAG7 fragmentation scheme that splits fields into 7-bit fragments and creates its own SIP7-SIP1 tables. For FRAG7, SIP7 contains both BVs and EVs, and the rest of the tables contain only EVs. Similarly, the DIP, SP and DP fields are fragmented to create the respective DIPi, SPi and DPi tables. As we fragment the remaining rules, the BV-EV vectors in the tables are appended and/or updated. We create distinct tables for the distinct groups G15-G1.

Pairing of fields: For every rule during pre-processing, we operate on all field pairs using both fragmentation schemes. The various pairings for the G15 group are: SIP-DIP, SIP-SP, SIP-DP, DIP-SP, DIP-DP, SP-DP. Similarly, for G3 there is only one pair SP-DP, and so on. Groups like G1, G2, G4 and G8 which contain

rules with just one valid field will be looked up directly using BV-EV and summation tuples. Now let us look at the pairing process with an example. For simplicity, we will consider the SIP and DIP fields for rules 1 and 3 only. Both rules belong to G15, thus their BVs and EVs for pairs of fields are stored in common tables.

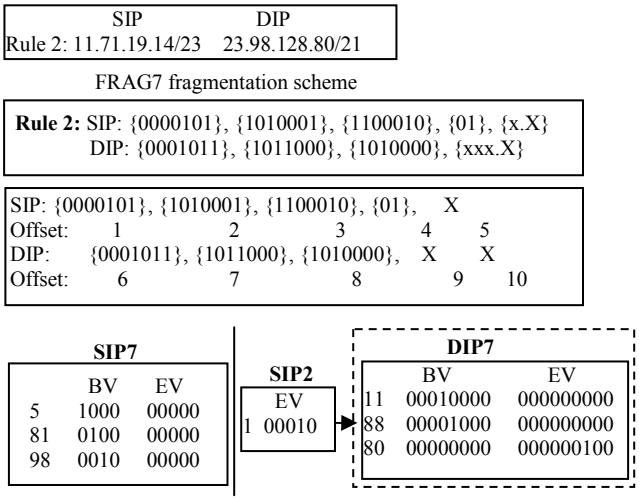

Figure 2. Pairings and BV-EV vector generation for the SIP and DIP fields using FRAG7.

The information regarding field pairing is stored in the SIP-DIP tables as follows. The BV and EV vectors for the first field, SIP, is generated as discussed before. The BV and EV vectors for the second field are calculated by combining the fields to get the correct offset values for the second one. For every table corresponding to the first field, there will be up to 8 tables under FRAG8 and 7 tables under FRAG7 that can contain information about the second field. This is because in a pair the first field can contain 1 to 8 fragments (for FRAG8); thus, the information regarding the second field in the pair is stored in the respective table set (i.e., the bit length with which the first field had ended). For the pairing of SIP and DIP, we denote the tables in the second field as SIP1-DIP to SIP8-DIP for FRAG8, and SIP1-DIP to SIP7-DIP for FRAG7; SIP1 indicates information (BV, EV) about the first field that ended in a 1-bit fragment and the SIP1-DIP tables contain information about the DIP field rules for which the first field (SIP) ended in a 1-bit fragment. In some cases the tables may be empty, thus not requiring any memory for storage. Fig. 2 shows the SIP and SIP-DIP tables for rules 1 and 3 under FRAG7. The first field of rule 1 ends in a 4-bit fragment and, hence, the DIP information of the second field in rule 1 is stored in the SIP4-DIP tables which are made up of DIP7 and DIP4 (since there are valid fragments of length 7 and 4 in DIP). Similarly, the first field in rule 2 ends with a fragment of 2 bits and, hence, the DIP information for rule 2 is stored in the SIP2-DIP tables. SIP7 does not have a corresponding DIP table because none of the two rules considered has a SIP field ending with 7 bits. Since we are illustrating FRAG7 in this figure, the BV vector of DIP in the second field can start at offset 2 and can go up to offset 9. Thus, the BV vector for the second field will contain 8 bits. Similarly, the EV offsets vary from 2 to 10 and, hence, will have 9 bits. As more rules are considered, we update the tables with the new information accordingly.

Port ranges: Since we represent the fields as patterns, the port ranges are represented in the prefix format. For example, if there is a rule with the SP specified as the range 0:1024, then this field is in binary 0000000000000000:0000010000000000, which is transformed into the two prefix patterns {000000xxxxxxxxxx} and {0000010000000000}. The procedure of creating the BVs and EVs under FRAG8 and FRAG7 is then carried out in the same way as previously shown. The pairing process on fields is also carried out as discussed above. This pairing process may result in the duplication of some rules that will result in multiple summation tuple lookups per rule (explained later on); however, the results section shows that the increase in memory consumption is not much.

Choice of fragmentation: We use two fragmentation schemes here because they collectively help us filter out false detection alerts during the verification phase that employs pre-calculated table lookups. Although fictitious matches may still be possible, most of them are normally filtered out.

The choice of 7 and 8 bits for simultaneous fragmentation was made based on experimental results which showed that any number of bits lesser than 7 for fragmentation increases the time in clock cycles for the next packet to be inputted into the system, which directly affects the throughput. On the other hand, any choice greater than 8 bits almost doubles the memory usage. Thus, the 8-bit and 7-bit fragmentations present a good tradeoff between throughput and memory usage.

Summation m-tuples: Once all of these operations have been performed on the rule-set, the tables containing the BVs and EVs will be available for every group. A separate block contains unique weights for field fragments ranging from 1 to 8 bits in length. There is one block per field and there are 8 different weight tables WT_1 to WT_8 in each block, as shown in Fig. 3. The weight block takes one byte of input for addressing and every weight table picks up the appropriate number of most significant bits of the byte. Every entry in the weight table is assigned a random m-tuple of weights represented by vector $W = \{weight_1, weight_2, \ldots, weight_m\}$; let $bw_1, bw_2, \ldots, bw_m$ be the respective number of bits in each weight element. The choice of the bws is such that we create unique weight m-tuples for the fragments in a table and also give some leeway for generating unique summation m-tuples for distinct rules. Since we assume 8-bit fragments, the bws should collectively be at least 8 bits to distinguish among 2^8 or 256 possible fragments; however, we reserve two additional bits in every table in order to simplify the weight assignment process and also to accommodate future updates. We conveniently use m=3 and 10 bits per m-tuple in the WT_8 table with the distribution of 4 bits, 3 bits and 3 bits for bw_1, bw_2 and bw_3, respectively. Any number of m greater than 3 needs more adders and comparators as we see later on.

Using these weight m-tuples we pre-calculate a *summation m-tuple* for each field in the rule using 8 bits per fragment; tails can have less than or equal to 8 bits. Consider rule 10 in Table I with the two valid fields {11.23.131.145/29; 0:1023}. As seen earlier a port range of 0:1023 turns into a pattern "000000X" with the most significant 6 bits being common per range and the rest being don't cares. We calculate the summation m-tuple for this rule at static time:

1) Split the fields into 8-bit fragments assuming FRAG8:
Field 1: "00001011" "00010111" "10000011" "10010"
Field 3: "000000"

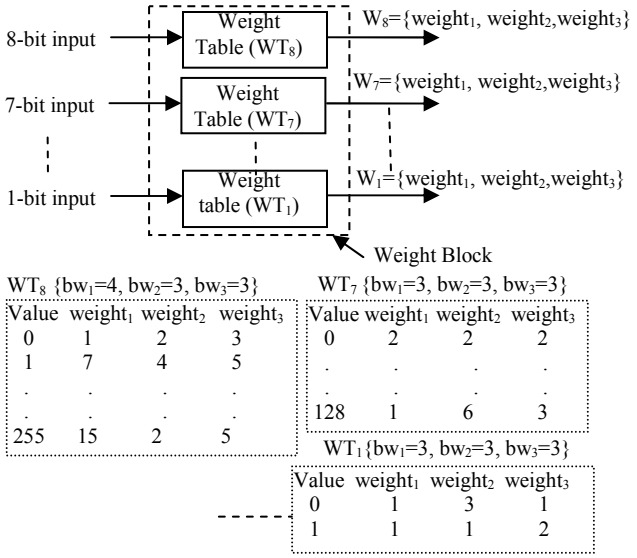

Figure 3. Weight tables for m=3.

2) To derive the summation m-tuples for these fields, apply the following position-weighted, element-wise summations involving the respective weight-tuples of the constituent fragments:
SUM(Field1) = WT_8("11") + 2 * WT_8 ("23") + 4 * WT_8 ("131") + 8 * WT_5("18"); SUM(Field3) = WT_6 ("0");
The weight m-tuple tables for distinct fields are generally different even though the m-tuple exclusivity principle is on per table basis.
3) Once we have the individual summation m-tuples per field, we sum them up along with the protocol value (in binary) of the rule to obtain the final summation m-tuple for the rule.
SUM(rule 10) = SUM(Field1) + SUM(Field3) + Protocol

This summation method is applied to all the rules. The summation m-tuples act as addresses for table lookups in relation to the incoming packet. Fig. 4 shows the chosen summation m-tuples for the rules in Table I (assuming m=3 and the weights in Fig. 3). Once we have pre-calculated the summation m-tuple of a rule, we store it along with the rule ID and the action to be taken for this rule in a table at a location which is produced by a hash function on this summation m-tuple. For our example, we create up to 16 distinct tables to deal with 16 groups of rules. However, since some rules of certain groups are lesser in population than others, we can combine storage for their summation m-tuples into one table instead of having separate tables for each of these groups. Also, if the protocol field of a rule is 'any,' which means that the rule must be triggered independent of the protocol, then we store the summation m-tuple without using the protocol field in a separate table that is accessed concurrently.

The assignment of weight m-tuples is done cautiously to generate summation m-tuples for the rules that can never result in false positives (explained later in Section 3.4). The weight block works independently of the FRAG blocks. Also, there is no pairing or grouping of fields in relation to this block. Its purpose is to generate summation m-tuples for all possible sub-patterns in each field, independent of their length in bits.

	Sum_1	Sum_2	Sum_3
Rule 1:	108	88	129
Rule 2:	175	121	144
Rule 3:	85	46	91
.	.	.	.

Figure 4. Summation m-tuples for the rules in Table 1

SUM= (Sum_1, Sum_2, Sum_3).

3.2 Runtime Rule Matching

The information about the position of individual fragments in a pair of fields appearing in a rule is encoded in the vector tables, as shown in Fig. 2. We also have the summation m-tuples for every rule stored in the tables. Now we have to devise a way to find out at runtime if fields in an incoming packet match the valid/important fields in any rule. This can be accomplished by using the BV and EV vectors in the SIP-DIP tables to create a lookup address; the latter will depend on the incoming packet header and will appropriately combine the retrieved weight m-tuples to access the summation tables. The incoming fields are first forwarded to the fragmentation-based tables of every group containing the BV and EV vectors and also to the weight tables containing the weight m-tuples. OR, AND and SHIFT operations are performed on the BV-EV vectors as discussed below.

3.2.1 Group Block

All possible pairings of rule-valid fields are considered for each group of rules. Hardware pairing blocks EQi perform the function of field detection using BV-EV tables; i=1,…,8 for FRAG8 and i=1,…,7 for FRAG7. Let us explain the operations in these blocks using the SIP-DIP pair (as shown in Fig. 5) and FRAG8; this process can be easily modified for other field pairs and FRAG7. EQ8 contains the DV(8) vector that detects the non-tail fragments of the pair in the incoming packet header and the EDV(8) vector that detects the tail fragment. DV(8) is initialized to "100…0" at the SIP side of the EQ8 block.

For the DIP block it is initialized to "01111000" since the source IP field in the rule can be of the form s1x.x.x.x , s1.s2x.x.x, s1.s2.s3x.x or s1.s2.s3.s4x, where s1, s2, s3 and s4 represent the prefix part of a source IP with a maximum of 8 bits in them; the second field in the SIP-DIP pair can start at the 2nd to 5th position in the offset vector. DV(8) has the same length as EV. The following equations are performed in block EQ8 ("&" represents concatenation and ">>" represents a right shift):
$DV(8)_{n+1}$= ($DV(8)_n$ AND (BV(8) & '0')) >> 1 ;
(DV(8) is present only for SIP8)
$EDV(8)_{n+1}$= $DV(8)_n$ AND EV(8) ;
where n is the current clock cycle. At every clock cycle, using a fragment of the input packet field the tables are accessed, and the corresponding BV and EV vectors are outputted into EQ8. Since the other tables contain only EVs, blocks EQ1 to EQ7 perform:
$EDV(i)_{n+1}$= ($DV(8)_n$ AND EV(i)) >> 1 ; for i= 1, 2, …,7.

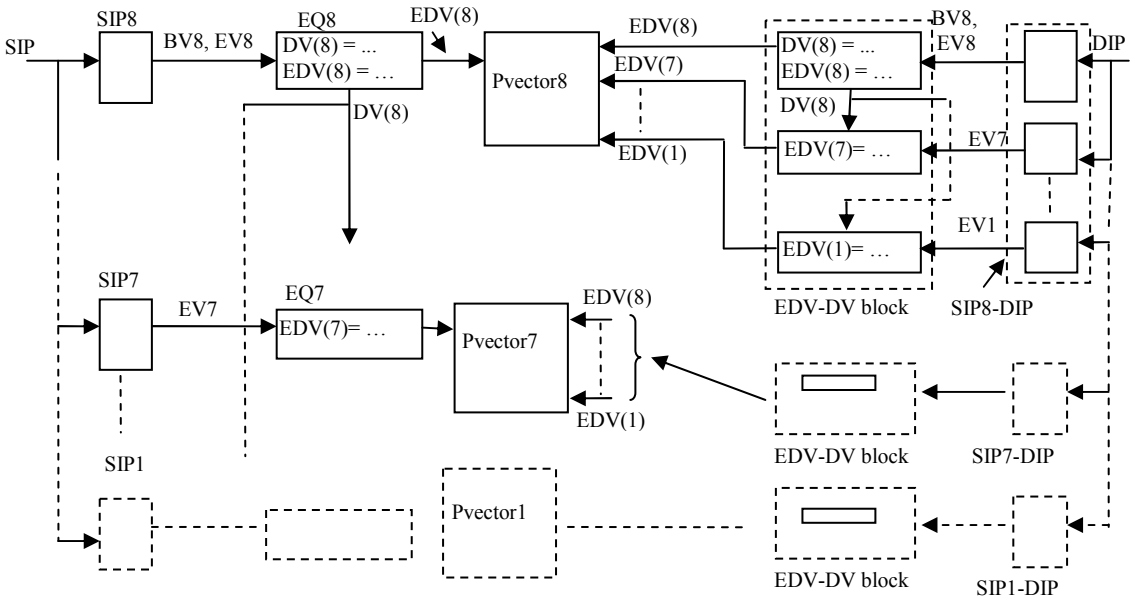

Figure 5. SIP-DIP block using FRAG8 (SIPDIP-FRAG8).

The Pvector8 block in Fig. 5 generates four 32-bit vectors for the SIP prefixes of 8, 16, 24 and 32 bits in the first to fourth clock cycles; they are denoted by P(8), P(16), P(24) and P(32), respectively. Likewise the Pvector7 block generates the vectors P(7), P(15), P(23) and P(31) in these four clocks cycles.

1st clock cycle: 11 comes into SIP and 23 comes into DIP

SIP: DV(8)= {"1000" AND ("100" & '0')} >> 1 = "0100" ;
DIP:DV(8)= {"0111100" AND ("00010000" & '0')} >> 1 = "00001000"

2nd clock cycle: 71 comes into SIP and 98 comes into DIP

SIP: DV(8)= {"0100" AND ("010" & '0')} >> 1 = "0010";
DIP: DV(8)= {"00001000" AND ("0000100" & '0')} >> 1 = "00000100"

3rd clock cycle: 71 comes into SIP; the most significant 7 bits are forwarded to SIP7; for DIP the most significant bits are forwarded to the DIP tables; however , the 5 most significant bits are the point of interest.

SIP: EDV(7)= "0010" AND "0010" = "0010" ;
DIP: EDV(5)= "00000100" AND "00000100" = "00000100";

EDV(5) is non-zero after 3 clock cycles and, hence, we right-shift EDV(7) by 3 and AND them together: {(EDV(7)}>>3 AND EDV(5) = {"0010" >> 3 } AND "00000100" = "00000100" AND "00000100" which is non-zero.
P(23)= "00000000000000000000100000000000";

Figure 6. Detection example using FRAG8.

Let's look at an example where a packet with source IP:11.71.19.14 and destination IP:23.98.128.80 arrives. We will show the operations taking place in the SIPDIP-FRAG8 block. It can be seen from the rule-set that rule 3 has the first two fields of interest in the incoming packet. Fig. 6 shows the operations taking place in the SIPDIP-FRAG8 block. EDV(7) is non-zero in the 3rd clock cycle which means that the SIP value with prefix length 23 was found. To make sure that the second field was also found, as the information stored is for field pairs, we look into the EDVs coming out of the DIP blocks. We notice that the EDV(5) value in the 3rd clock cycle is non-zero, implying that we have to right shift the non-zero EDV of SIP by 3 bits and AND it with EDV(5). This operation results in a non-zero vector. Thus, we can represent the match by having the 21st bit of the P(23) vector as non-zero. The same set of operations are applied in the SIPDIP-FRAG7 block; by ANDing the two vectors we see that P(23) is non-zero with the 21st bit set to '1'.

Let's look at how the pairing of blocks in individual groups is arranged to make a decision about the complete rule. Let us take Group G13 with rule-valid fields SIP, DIP and DP. The targeted table blocks will be SIP-DIP, SIP-DP and DIP-DP. We need the FRAG8 and FRAG7 fragmentation schemes for the SIP-DIP tables. Just one scheme for SIP-DP and DIP-DP is sufficient because SIP8-DIP8 (FRAG8) and SIP7-DIP7 (FRAG7) represent SIP and DIP using both schemes; we can optimize the representation of DP7 and DP8 using SIP8-DP8 and DIP7-DP7, or SIP7-DP7 and DIP8-DP8. Fig. 7 shows the block diagram for the G13 field pairings.

The intersection blocks stemming from the SIP-DIP blocks perform the AND operation on the vectors coming out of SIPDIP-FRAG8 and SIPDIP-FRAG7. The intersection block for SIP-DIP-DP finds the common bits in the three fields and forwards the values to the address generation block (described below) in order to select the summation m-tuples generated by the summation block. For example, if a packet header matches in the FRAG blocks 24 prefix bits of SIP, 23 and 27 prefix bits of DIP simultaneously, and 16 prefix bits of DP, then the intersection blocks will forward two 14-bit vectors (since a SIP or DIP prefix length can be encoded with 5 bits each, and the DP prefix length can be encoded with 4 bits) to the address generation block. The first vector will be the concatenation of ("10111", "10110", "1111") for SIP=24, DIP=23 and DP=16 and the second vector will be the concatenation of ("10111", "11010", "1111") for SIP=24, DIP=27 and DP=16.

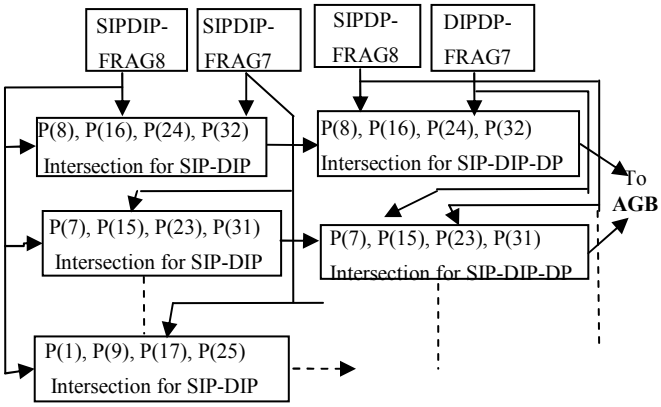

Figure 7. Pairings for Group G13.

3.2.2 Summation Block

This block contains adders and shifters. It pre-calculates the summation m-tuples in parallel while the BV and EV calculations are going on in the group blocks, and stores them in temporary registers (SUM BANK block) in the various groups. Fig. 8 shows the summation block.

3.2.3 Detection Block

The detection of a rule in a packet header is performed by combining the processes of the group block, summation block and address generation block.

Address Generation Block: It consists of the SUMF and SUMT blocks. Fig. 9 shows the address generation block for G13. The vector obtained from the intersection blocks is used to access the summation values from the SUM BANKs, which are added and hashed to access the summation m-tuples. The summation m-tuples table (SUMT) contains the summation m-tuples, the rule ID and the action to be taken on the packet. Every group forwards its best rule ID and eventually the best rule ID from all the groups is used to take action on the packet. Since the rules are ordered (i.e., the lowest ID represents the best matching), we know which one to select. One of the summation m-tuples table is accessed directly without the protocol added to the summation, whereas the other table is accessed with the protocol field added.

The concentration of rules is such that some groups will have more rules than others. To reduce storage, we simultaneously accommodate some rule groups into common summation m-tuple tables. The final block diagram of the architecture for packet classification is shown in Fig. 10 where we have clubbed together G1-G12 into one block as their small populations do not warrant separate SUMF blocks (this is just an example to show that the blocks can be clubbed together).

3.3. Rule Splitting Method

We place the summation m-tuples in the RAM tables in such a way that avoids collisions. This is ensured by appropriately adjusting the weight m-tuples at static time and choosing the bit widths of individual weight elements. We used in our experiments up to four RAMs in a group that can be accessed in parallel to check for matching m-tuples. The summation m-tuples are placed in the RAMs at locations which are produced by a hash function on the summation m-tuples. The hash functions apply simple XOR and modulo add functions which can be easily implemented in hardware. In cases where it is very difficult to place new m-tuples in non-vacant locations, we apply a rule splitting method at static time. The data structure stored in this case is slightly different than the standard one. For example, consider the rule (SIP: "123.4.5.89"; DIP: "135.6.7.0/23"; SP: 0:65535; DP: 1023:1023; Protocol: 6) with rule ID 1000. This rule falls into the G13 group. If this rule cannot be placed into a vacant RAM location, then we split the rule into two or more parts.

For example, the aforementioned rule can be broken into (SIP: "123.4.5.89"; DIP: "135.6.7.0/23") for which the summation m-tuple is stored in the G12 group, if there is no collision, and (DP:1023) which belongs to the G1 group. We also store information about this split in order to combine at run time the individual part matches. For the above example, the summation m-tuples in the G12 RAM will be stored along with the rule ID and the group number of the second part of the rule (which is G1) as well as the new rule ID (which is assigned to the second part during the split). This splitting method can also be used to control the number of 1's in Pvector to optimize pipeline scheduling, thereby having a positive effect on the throughput. We could easily space them out into different groups by using a constraint on the number of 1's in a Pvector at any given time. This is done by generating a binary tree and checking the number of rules which can reflect a '1' in Pvector. We already have different SUMT blocks for different sets of Pvectors, which makes it better since we do not encounter more than five 1's in Pvectors for the rule set which we implemented.

3.4. Elimination of False Positives

The weight assignment process makes sure that any two genuine rules have different summation m-tuples. A false positive is then possible only if a fictitious combination of fragments (where at least one fragment is not part of the rule but is from a different rule) causes a non-zero EDV and also generates a summation m-tuple which is the same as that of a genuine rule. During pre-processing the population of rules in a group is further sub-divided and rules are clubbed together based on the length of the prefix in the field or fields to create an even distribution of summation m-tuples in the RAMs. Let us look at an example with a set of rules whose summation m-tuples are placed in the same set of RAMs. Let us investigate the possibility of false positive generation for two of the rules:

	SIP	DIP	SP	DP
1	97.166.41.112/32	182.125.194.192/32	0:65535	0:65535
2	27.26.30.130/32	127.206.10.2/32	0:65535	0:65535
3	……	…..	….	…

Assume an incoming packet with (SIP: 97.166.41.112) and (DIP: 127.206.10.2). This packet contains an SIP that matches the 1st rule and a DIP that matches the 2nd rule. The position-based encoding of the bit vectors for the two rules will produce a non-zero EDV because the header fragments will match partially more than one rule. If the summation m-tuple generated by this header is equal to some genuine summation m-tuple in the same set of RAMs, then the latter rule will be triggered. Thus, during pre-processing we check for all possibilities of such fictitious non-zero EDVs and ensure that they do not generate a genuine summation m-tuple that will cause a false rule to be triggered.

This is accomplished by either tweaking the weight m-tuples or applying the aforementioned rule splitting method. However, the majority of such cases are filtered out because of using simultaneously two fragmentation schemes (FRAG7 and FRAG8); a packet may match an incorrect rule under a scheme but it does not often match any rule under the other scheme.

4. Experimental Results

Classbench [21] was used to generate rule-sets. Optimizations were performed in the implementation stage; for example, if a particular table had less than 8 members, then the use of RAM was avoided and only comparators were used instead (to save on memory). If a particular group, say G15, had very few rules (less than 100), then the rule splitting method was used to separate the rules instead of allocating a distinct set of blocks for them. Fig. 11 shows the number of BRAMs (Xilinx FPGA Block RAMs) needed for various numbers of rules. The BRAM consumption is only for GROUP blocks that store bit vectors and not for SUMT blocks. We can deduce that the memory needed to store bit vectors flattens as we increase substantially the number of rules, thus indicating that the memory needed for packet classification grows linearly with the number of rules as we still need to store distinct summation m-tuples for distinct rules.

The design is pipelined and has a worst case latency of 23 clock cycles. It takes 5 clock cycles to retrieve the EV-BV pair and 3 clock cycles for the DV-EDV calculations. We created a decision tree for the rules and found out that there could be a maximum of five ones in a Pvector; however, the real number is 3 since we separate Pvectors into different SUMT blocks of summation tables which are independent of each other based on prefix lengths. Still considering 5 as the worst case number, we will then need 7 clock cycles (5 + 2 FIFO latency) for the last non-zero Pvector to go into the summation selection block SUMF. This is then followed by add operations, hashing and RAM access, which take another 5 clock cycles. The final comparison of rule IDs takes 3 clock cycles. Thus, the worst-case latency for rule classification is 23 cycles. The Group, SUMT and SUMF blocks work independently of each other and are interfaced with appropriately sized FIFOs in their input. When a block is working on a packet header, the other blocks can work on a different packet header. Thus a new packet can be inputted into the design every 8 clock cycles.

Table 2. Results for various Classbench-produced files

File	Rules	Rules due to port duplication	BRAMs
acl1	10,246	13,841	111
acl2	10,553	20,144	141
acl3	10,205	17,106	133
acl4	10,107	16,184	127
acl5	8123	10,433	108

Table 2 shows the memory consumption for various rule-sets that were created by Classbench. The memory consumption is for the entire design. An example of port duplication is a rule with port range 0:1024 that will generate two summation m-tuples, one for the port range 0:1023 represented by "000000x" and the second for 1024 represented with "0000010000000000". The hardware design was implemented using VHDL and was synthesized using Synplify Pro. We implemented the design for the acl3 file of Classbench on a Xilinx Virtex II Pro XC2VP70 FPGA, consuming 43,487 logic cells, 46,450 flip-flops and 128 BRAMs. Only 151 Kbytes are used to store the rules and the bit vectors. 4, 3 and 3 bits for the first, second and third weight tuple were used, respectively, for a total of 10 bits per weight triplet. Also, a maximum of 9, 7 and 7 bits were used to represent individual summation triplets, respectively, for a total of 23 bits in the summation triplets.

Table 3 shows a comprehensive comparison of our design with others. Our design runs at 242.1 MHz, which represents the highest operating frequency. The throughput in the worst case (assuming 40 bytes per packet) is 9.68 Gbps. Our design has the lowest memory consumption even with more than 10,000 rules. In relation to [2], we notice that although our design performs around 8 times slower, [2] uses dual-port memories with 407 Virtex-5 BRAMs which come in 36Kbit blocks. Since the Virtex-II Pro BRAMs come in 18Kbit blocks, [2] represents 814 18Kbit blocks whereas our entire design uses only 128 such BRAMs. Memory in our design is present in the Group and SUMT blocks. As the number of rules grows to more than 10,000, the memory curve for the Group blocks starts flattening out (Fig. 11) implying that the memory needed to store BVs and EVs does not grow when adding new rules. Since additional memory is then consumed only by new summation m-tuples, the memory increases only linearly with the rule-set population.

5. Conclusions

We have presented a novel method for packet classification which has the highest operating frequency among all known designs, and has a good balance between throughput and memory usage for more than 10,000 rules. We have also shown that the memory consumption grows only linearly with the number of rules. We could make our design input a new field per clock cycle, which could further improve its throughput. This will become a future research objective. Our scalable design can easily be extended to achieve a higher than 10Gbps throughput by employing multiple FPGAs running in parallel and storing different rules.

Table 3. Comparison with other works

	FPGA Device	*Frequenc MHz*	*Rules*	*Memory (Kbytes)*	*Gbps*
[2]	Virtex-5	125 MHz	9603	612	80.23
[3]	Cyclone-3	128 MHz	10,000	286	3.41
[5]	Virtex-4	153 MHz	4000	178	1.88
[4]	Virtex II Pro	N.A.	128	221	16
Ours	Virtex II Pro	242.1MH	10,205	151	9.68

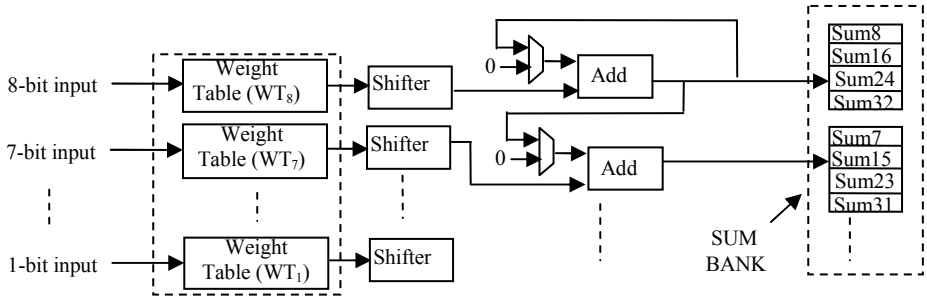

Figure 8. Summation block.

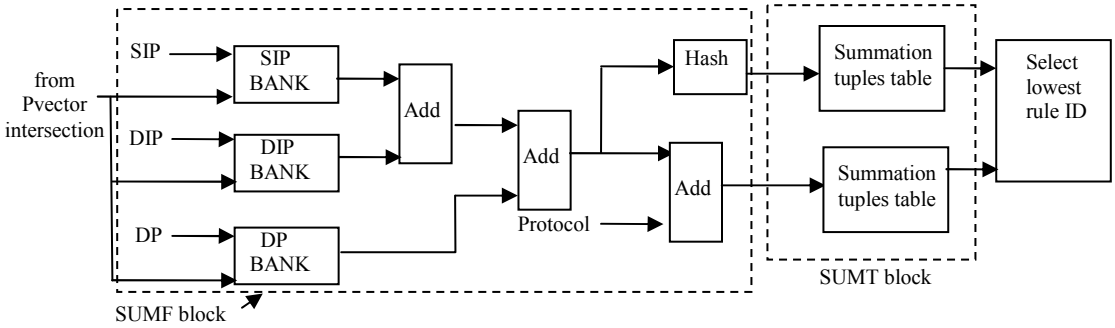

Figure 9. Address generation block for Group G13.

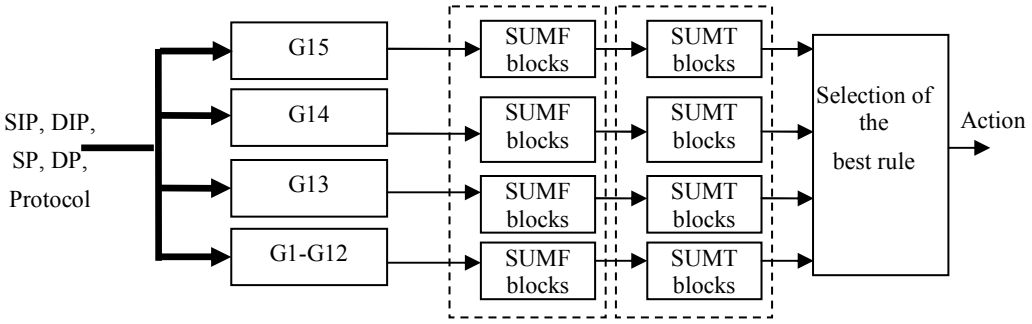

Figure 10. Block diagram for ultimate detection.

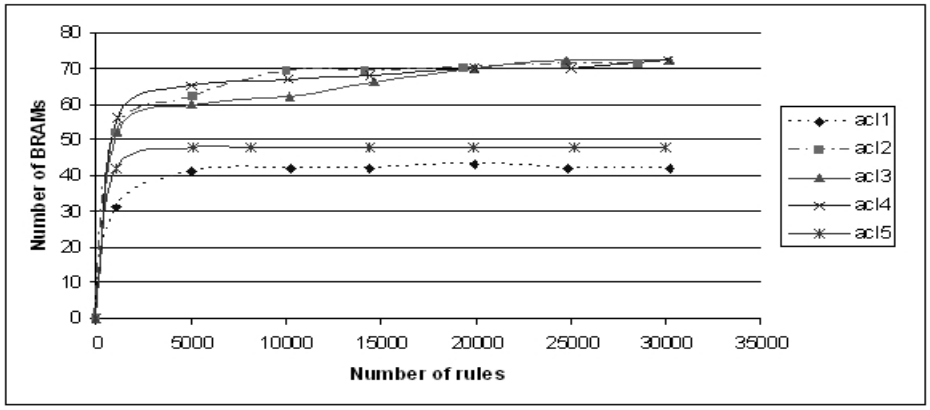

Figure 11. Number of BRAMs consumed by group blocks as a function of the rule population.

References

[1] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," Hot Intercon. VII, Aug. 1999.

[2] W. Jiang and V. Prasanna, "Large-scale wire-speed packet classification on FPGAs," ACM/SIGDA Symp. FPGAs, 2009

[3] A. Kennedy, X. Wang, Z. Liu, and B. Liu, "Low power architecture for high speed packet classification," Arch. Network Commun. Systems, 2008.

[4] G.S. Jedhe, A. Ramamoorthy, and K. Varghese, "A scalable high throughput firewall in FPGA," FCCM Conf., 2008.

[5] A. Nikitakis and I. Papaefstathiou, "A memory-efficient FPGA-based classification engine," FCCM Conf, 2008.

[6] D.E. Taylor and J.S. Turner, "Scalable packet classification using distributed crossproducing of field labels," INFOCOM, 2005.

[7] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," SIGCOMM, 2003.

[8] S. Dharmapurikar, H. Song, J.S. Turner, and J. W. Lockwood, "Fast packet classification using bloom filters," Architectures for Network and Communications Systems, pages 61–70, 2006.

[9] I. Papaefstathiou and V. Papaefstathiou, "Memory-efficient 5D packet classification at 40 Gbps," INFOCOM, 2007.

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

[11] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: hardware/software IP lookups with incremental updates," In SIGCOMM Comput. Commun. Rev., 34(2):97–122, 2004.

[12] F. Baboescu and G. Varghese, "Scalable packet classification," in Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, pages 199–210, 2001.

[13] T.V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," in Proceedings of ACM SIGCOMM, pages 191-202, September 1998.

[14] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," in Proc. SIGCOMM, pages 193–204, 2005.

[15] F. Yu, R.H. Katz, and T.V. Lakshman, "Efficient multimatch packet classification and lookup with TCAM," in IEEE Micro, 25(1):50–59, 2005.

[16] M.H. Overmars and A.F. Van der Stappen, "Range searching and point location among fat objects," in Journal of Algorithms, 21(3), pp. 629-656, November 1996.

[17] P. Gupta and N. McKeown, "Algorithms for packet classification," in IEEE Network, vol. 15, pp. 24–32, Mar/Apr 2001.

[18] D.E. Taylor, "Survey and taxonomy of packet classification techniques," in ACM Comput. Surv., vol. 37, no. 3, pp. 238–275, 2005.

[19] P. Tsuchiya. "A search algorithm for table entries with non-contiguous wildcarding," unpublished report, Bellcore.

[20] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. "Fast and Scalable Layer Four Switching," in Proc. of ACM Sigcomm, pages 203-14, September 1998.

[21] D. Taylor and J. Turner, "ClassBench: A Packet Classification Benchmark", in IEEE Infocom'05, March 2005.

[22] X. Wang and S.G. Ziavras, "Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines," Concurrency Computation, Vol. 16, April 2004, pp. 319-343.