Journal of Circuits, Systems, and Computers
Vol. 28, No. 14 (2019) 1950237 (26 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0218126619502372

AFBV: A High-Performance Network Flow Classification Method for Multi-Dimensional Fields and FPGA Implementation*

Ling Zheng[†], Zhiliang Qiu[†], Weina Wang[†], Weitao Pan[†,¶], Shiyong Sun[‡] and Ya Gao[§]

[†]*State Key Laboratory of Integrated Services Networks, Xidian University, Xi'an, Shaanxi 710071, P. R. China*

[‡]*Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory, The 54th Research Institute of China Electronics Technology Group Corporation, Shijiazhuang, Hebei 050081, P. R. China*

[§]*School of Internet of Things Technology, Wuxi Institute of Technology, Wuxi, Jiangsu 214121, P. R. China*
[¶]*wtpan@mail.xidian.edu.cn*

Received 26 April 2018
Accepted 10 January 2019
Published 21 February 2019

Network flow classification is a key function in high-speed switches and routers. It directly determines the performance of network devices. With the development of the Internet and various kinds of applications, the flow classification needs to support multi-dimensional fields, large rule sets, and sustain a high throughput. Software-based classification cannot meet the performance requirement as high as 100 Gbps. FPGA-based flow classification methods can achieve a very high throughput. However, the range matching is still challenging. For this, this paper proposes a range supported bit vector (RSBV) method. First, the characteristic of range matching is analyzed, then the rules are pre-encoded and stored in memory. Second, the fields of an input packet header are used as addresses to read the memory, and the result of range matching is derived through pipelined Boolean operations. On this basis, bit vector for any types of fields (AFBV) is further proposed, which supports the flow classification for multi-dimensional fields efficiently, including exact matching, longest prefix matching, range matching, and arbitrary wildcard matching. The proposed methods are implemented in FPGA platform. Through a two-dimensional pipeline architecture, the AFBV can operate at a high clock frequency and can achieve a processing speed of more than 100 Gbps. Simulation results show that for a rule set of 512-bit width and 1 k rules, the AFBV can achieve a throughput of 520 million packets per second (MPPS). The performance is improved by 44% compared with

*This paper was recommended by Regional Editor Tongquan Wei.
¶Corresponding author.

FSBV and 30% compared with Stride BV. The power consumption is reduced by about 43% compared with TCAM solution.

Keywords: Flow classification; switching system; range matching; field programmable gate array (FPGA); pipeline architecture.

1. Introduction

Network flow classification technology is a key function in high-speed switches and routers. It is widely used in policy-based routing, quality of service (QoS), network load balancing, network security, and many other fields.[1–3] It directly determines the performance of network switching and routing equipment. The objective of flow classification is to classify the input packets into different network flows according to the packets' headers and a given rule set. Then the packets are processed according to the matched rules.[4]

Recently, with the fast development of Internet and various kinds of applications, the flow classification is becoming more and more challenging. It needs to support multi-dimensional fields, large rule sets, and sustain a high throughput. First, the dimension of flow classification is increasing. The number of match fields is developing from 5-tuple to 44 fields that OpenFlow[5] 1.5 specifies. Then the number of fields will keep growing with the development of programmable data plane.[6,7] Second, the increasing amount of match fields leads to the rapid growth of the number of rules. The size of traditional L2 or L3 table is usually thousands of entries, but the size of the flow table in software-defined networking (SDN).[8,9] switch is up to 4 million entries.[10] Besides, the matching algorithms of different fields are more diverse and more complex. L2 and L3 lookups use exact matching and longest prefix matching; while the flow classification not only needs to perform exact and longest prefix matching, but also needs to support arbitrary wildcard matching and range matching. Finally, with the appearance of 40/100GE standard, the ever-increasing bandwidth puts a higher demand on the processing speed and real-time performance. Therefore, the high-speed flow classification technology has attracted more and more attention in both industry and academy.

The existing flow classification algorithms can be classified into the following categories.

(1) Algorithms based on ternary content-addressable memory (TCAM)[11–13]: TCAM can support arbitrary wildcard matching by saving bit-level masks,[14] and has a fast lookup speed of 400 million packets per second (MPPS). However, TCAM is quite limited in practical applications due to its large power consumption and high cost.[15] In addition, TCAM inevitably requires range expansion when storing range fields. A typical prefix expansion method[16] converts one rule into a number of prefixes. In the worst case, the number of rules after expansion will increase exponentially with the number of fields, resulting in a significant reduction in TCAM space utilization. Therefore, TCAM is unsuitable for range matching.

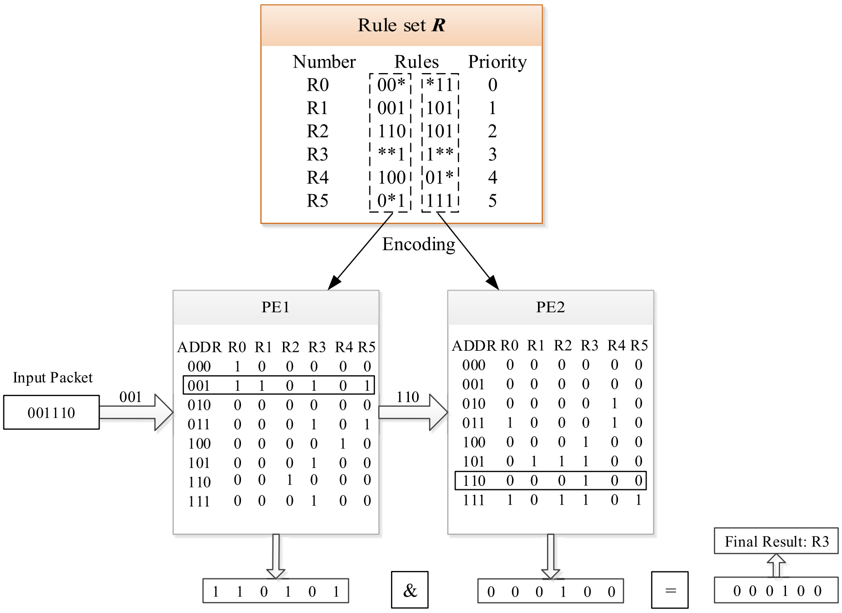
(2) Algorithms based on decision tree[17]: The basic idea of these approaches is to represent the rule set using a decision tree data structure, so that the search space can be reduced to smaller subspaces. The typical algorithms are binary trie,[18] H-Tree,[19] G-Tree,[20] etc. The tree data structure is a common method in routing lookup because it can conveniently represent the address prefixes.[21,22] However, when it comes to multi-field matching, the performance of decision-tree-based approaches is rule-set-dependent. A cut in one field can lead to duplicated rules in other fields (rule set expansion). Besides, the implementation of decision tree data structure requires expensive hardware cost.

(3) Algorithms based on space division.[23] Typical approaches include HiCuts,[24] HyperCuts,[25] EffiCuts,[26] etc. The HiCuts algorithm represents the rules in the d-dimensional space as a hyper-rectangle and the total search space is decomposed into multiple smaller subspaces. After determining the subspace, the final match result can be obtained through linear search. Since the implementations are more complicated, space-division-based algorithms are mainly implemented by software running on CPU or GPU,[27] so that the processing speed is much slower compared with hardware-based algorithms. It is difficult to achieve line-rate processing at a speed of 100 Gbps.

(4) Algorithms based on bit vector (BV)[28]: These methods use BVs to store the encoded rule set. When performing rule matching, the corresponding field of the input packet header is used as an address to read the BV matrix. The BVs are then extracted and bitwise "AND" operation is used to merge all the extracted BVs to generate the final match result. These approaches are usually implemented by field programmable gate array (FPGA) to achieve high processing speed. However, the required memory and logic resources are proportional to the bit width of rules (denoted as L) and the number of rules (denoted as N). When the size of the rule set is large, the long internal wires will seriously degrade the clock frequency.

To reduce the width of the internal signal, methods such as Stride BV[29] and Scalable BV[30] are proposed. The Stride BV splits an L-bit rule to $s = \lceil L/l \rceil$ sub-rules. This operation is called *Stride*, and l is called Stride length. The s sub-rules are encoded and stored, respectively. Thus, the number of extracted BVs is reduced from L to l. Figure 1 shows an example of Stride BV when $L = 6$, $l = 3$. When an input packet header is 001110, the first three bits match the rules R_0, R_1, R_3 and R_5, and the last three bits match the rule R_3. Through the logic "AND" operation, the final result is matching R_3. In Fig. 1, the module which processes a subrule set is called a processing element (PE) and symbol "$*$" means wildcard.

The BV-based methods support arbitrary wildcard matching and can achieve a very high throughput through pipeline-based FPGA implementation.[31] However, the main shortcoming is that matching errors may occur when BV-based methods are used to match the range fields. Reference 32 proposes an encoding scheme called range bit vector encoding (RBVE) to process range fields. But as mentioned above, internal wires of length $O(N)$ are used for the memory. As N increases, the clock rate



Number	Rules		Priority
R0	00*	*11	0
R1	001	101	1
R2	110	101	2
R3	**1	1**	3
R4	100	01*	4
R5	0*1	111	5

PE1

ADDR	R0	R1	R2	R3	R4	R5
000	1	0	0	0	0	0
001	1	1	0	1	0	1
010	0	0	0	0	0	0
011	0	0	0	1	0	1
100	0	0	0	0	1	0
101	0	0	0	1	0	0
110	0	0	1	0	0	0
111	0	0	0	1	0	0

PE2

ADDR	R0	R1	R2	R3	R4	R5
000	0	0	0	0	0	0
001	0	0	0	0	0	0
010	0	0	0	0	1	0
011	1	0	0	0	1	0
100	0	0	0	1	0	0
101	0	1	1	1	0	0
110	0	0	0	1	0	0
111	1	0	1	1	0	1

Fig. 1. An example of Stride BV method.

of the entire pipeline will deteriorate. Therefore, the BV-based methods lack effective support for range matching.

To meet the requirement of line-rate processing while effectively supporting range matching, this paper first proposes a range supported flow classification method called range supported bit vector (RSBV). Then the bit vector for any types of fields (AFBV) is proposed. The main contributions of this paper are as follows:

(1) Disadvantages of existing flow classification methods are analyzed, and it is pointed out that the existing methods cannot support range matching effectively while guaranteeing line-rate processing. To solve the problem above, the RSBV method is proposed. The procedure of the method is described in detail in terms of encoding, storage, lookup, and implementation architecture.
(2) On the basis of RSBV, a high-speed flow classification method for any types of fields, AFBV, is proposed. The fields of an input packet header are divided into range fields, and other fields (other fields include exact matching fields, longest prefix matching fields, and wildcard matching fields). The range fields are processed through RSBV, and other fields are processed by two-dimensional Stride BV (2D BV for short). The lookup results of the two types of fields are operated by logic "AND" operation and the final results can be obtained. The AFBV can support the flow classification for multi-dimensional fields including range matching. Through the pipeline-based implementation architecture, a throughput of beyond 100 Gbps can be achieved.

(3) The proposed algorithms are implemented and verified in Xilinx Virtex 7 xc7vx690t FPGA platform, which provides a reference design for the flow classification module of high-speed switches and routers. Experimental results show that the proposed method can sustain a throughput of 520 MPPS for a 1 k 13-tuple rule set with 512-bit width.

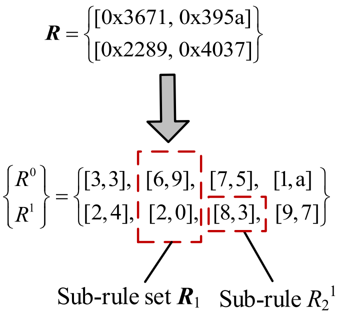
The rest of the paper is organized as follows. In Sec. 2, we describe the design and implementation of RSBV method in detail, and the performance of RSBV is analyzed. In Sec. 3, the AFBV method is introduced. Section 4 provides the experimental results. Finally, the conclusions are given in Sec. 5.

2. Range Supported Flow Classification Method: RSBV

2.1. *Symbol notation*

First of all, we give the following symbol notations. Let $\boldsymbol{R} = \{R^0, R^1, \ldots, R^{N-1}\}$ denote the range matching rule set. There are F fields in the rule set, and the bit width of each field is L. The amount of rules in $\boldsymbol{R}$ is N. The rules in $\boldsymbol{R}$ are sorted in descending order according to their priority. That is $\forall i < j$, $\text{pri}(R^i) > \text{pri}(R^j)$. In the case of $F = 1$, let the jth rule be $R^j = [L^j, U^j]$, where L^j is the lower bound and U^j is the upper bound. Let the corresponding field value of the input packet header be K. If $L^j \le K \le U^j$, then the input packet matches rule R^j.

We use *Stride* technique[29] to split the rule set. Let the Stride length be l, then $\boldsymbol{R}$ is split into s subrule sets $\boldsymbol{R_0}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_{s-1}}$ with bit width l, where $s = \lceil L/l \rceil$. We call $\boldsymbol{R_k}$ the k-stage subrule set. Then R^j can be expressed by $R^j = \{[L_0^j, U_0^j], [L_1^j, U_1^j], \ldots, [L_{s-1}^j, U_{s-1}^j]\}$, where $R_k^j = [L_k^j, U_k^j]$ represents the jth subrule in the kth stage subrule set. A simple example of *Stride* for range matching is shown in Fig. 2, where $F = 1, N = 2, L = 16$, and $l = 4$. The data format in Fig. 2 is hexadecimal.



R = {[0x3671, 0x395a], [0x2289, 0x4037]}
{R⁰, R¹} = {[3,3], [6,9], [7,5], [1,a]; [2,4], [2,0], [8,3], [9,7]}
Sub-rule set R₁
Sub-rule R₂¹

Fig. 2. A simple example of *Stride* for range matching.

2.2. Algorithm description of RSBV

The RSBV algorithm includes the following three main steps: encoding, storage, and lookup. In this subsection, we describe the design RSBV algorithm in detail. For simplicity and clarity, in this subsection, we assume $F = 1$.

2.2.1. *Encoding*

In Stride BV, only one bit is used to express one rule, due to there being only two situations: match or miss. But in range matching, there are multiple situations such as greater than, less than, and equal to. Besides, different stages may correspond to different situations. Therefore, the encoding scheme in RSBV is more complex than Stride BV. For the sake of simplicity and without loss of generality, we discuss the range matching process by judging whether a three-digit number falls into a certain range, and thus obtain the correct encoding scheme. Let $K_0K_1K_2$ denote a three-digit number, and a range matching rule is $[L_0L_1L_2, U_0U_1U_2]$. After *Stride*, the first stage subrule is $[L_0, U_0]$, second stage is $[L_1, U_1]$, and the last stage is $[L_2, U_2]$. The judgement process is shown by a flowchart in Fig. 3. From the flowchart, we can observe that the output of each stage (except for the first stage) depends not only on the match result of the current stage, but also on the output of the previous stage. If we obtain the output of the previous stage and the match result of the current stage, we can uniquely determine the output of the current stage.

Based on the analysis above, we can exhaust all the cases that may occur in the process of range matching, and each case can be expressed by a unique code. During the matching process, the output of each stage is determined by the output code of the previous stage and the code of the current stage. We repeat the process above from the first stage until the final stage and we can obtain the final match result from the output of the final stage.

In addition, the analysis above is performed for a three-digit number $K_0K_1K_2$, and the analysis can be extended to arbitrary s-digit number $K_0K_1 \cdots K_{s-1}$. The judgment on K_0 conforms to stage 1, the judgment from K_1 to K_{s-2} conforms to stage 2, and the judgment on K_{s-1} conforms to stage 3. Therefore, we can divide the subrule sets $\boldsymbol{R_0}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_{s-1}}$ into three stages: the first stage, the middle stage, and the final stage. Let $R = [L, U]$ denote a certain rule. It is split into s subrules $\{[L_0, U_0], \ldots, [L_{s-1}, U_{s-1}]\}$ using *Stride*. When the input field value is $K = K_0K_1 \cdots K_{s-1}$, all the possible cases in the range matching and the encoding for each case are listed in Table 1.

Discussion. During the encoding, each bit has its own physical meaning. Let $a_k[m]$ denote the mth bit of the code at the kth stage, The meanings of the bits of the codes are summarized in Table 2. It can be found that each subrule requires 2–4 bits for encoding, depending on the stage in which it is located.

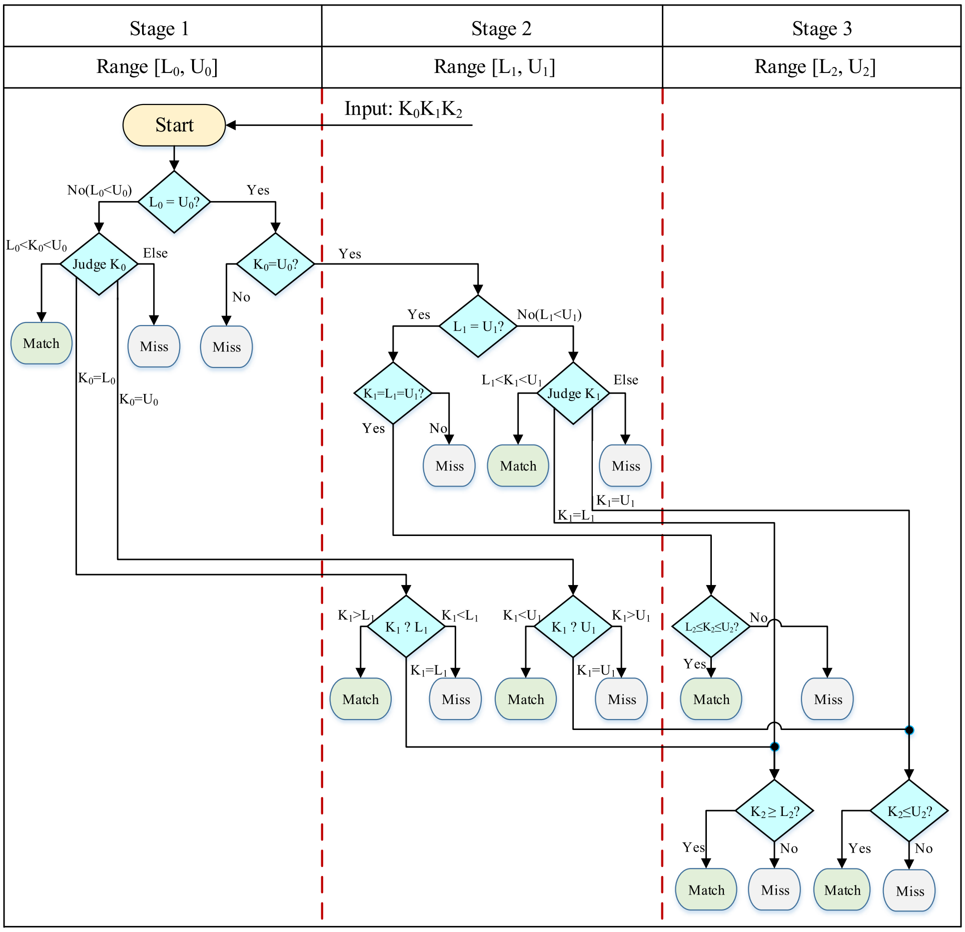


Fig. 3. Flowchart of the procedure of range matching for a three-digit number.

Table 1. All the possible cases in the range matching and the encoding.

First stage $k = 0$		Middle stage $1 \le k < s-1$		Final stage $k = s-1$	
Possible cases	Code	Possible cases	Code	Possible cases	Code
$K_0 = L_0 \neq U_0$	001	$K_k = L_k \geq U_k$	0010	$K_{s-1} \geq L_{s-1}$	01
$L_0 < K_0 < U_0$	010	$L_k < K_k < U_k$	0101	$K_{s-1} \leq U_{s-1}$	10
$K_0 = U_0 \neq L_0$	100	$K_k = U_k \leq L_k$	1000	$L_{s-1} \leq K_{s-1} \leq U_{s-1}$	11
$K_0 = L_0 = U_0$	101	$K_k = L_k = U_k$	1010	else	00
else	000	$K_k > L_k$ and $K_k > U_k$	0001		
		$K_k < U_k$ and $K_k < L_k$	0100		
		$K_k = L_k \leq U_k$	0110		
		$K_k = U_k \geq L_k$	1001		
		else	0000		

Table 2. The physical meaning of each bit of the codes.

Stage	Code	Physical meaning
$k = 0$ (First stage)	$a_0[0] = 1$	$K_0 = U_0$
	$a_0[1] = 1$	$L_0 < K_0 < U_0$
	$a_0[2] = 1$	$K_0 = L_0$
$1 \leq k < s-1$ (Middle stage)	$a_k[0] = 1$	$K_k = U_k$
	$a_k[1] = 1$	$K_k < U_k$
	$a_k[2] = 1$	$K_k = L_k$
	$a_k[3] = 1$	$K_k > L_k$
$k = s-1$ (Final stage)	$a_{s-1}[0] = 1$	$K_{s-1} \leq U_{s-1}$
	$a_{s-1}[1] = 1$	$K_{s-1} \geq L_{s-1}$

2.2.2. *Storage*

After encoding, the codes of the subrules need to be stored in the memory, according to the relationship of the addresses and rules. For subrule sets $\boldsymbol{R_0}, \boldsymbol{R_1}, \ldots, \boldsymbol{R_{s-1}}$, every subrule needs to be encoded, and after that, a three-dimensional matrix $\boldsymbol{D}[s \times N \times 2^l]$ is obtained. The element $D[k][j][i]$ is a $\{0, 1\}$ string with a bit width of e_k, and its value depends on the conditions described in Table 1. The meaning of $D[k][j][i]$ is the relationship of the input subfield value K_k and the subrule R_k^j when $K_k = i$. The total memory space for rule encoding is $2^l \times N \times \sum_{k=0}^{s-1} e_k$. The process of rule encoding and storage is described in Algorithm 1. Take the rule set shown in Fig. 2 as an example. The encoding result is shown in Fig. 4.

2.2.3. *Lookup*

During the lookup, the input field value K is split into subfields $K_0 K_1 \ldots K_{s-1}$, each of which has a bit width of l. In the kth stage, RSBV uses K_k as an address to read the matrix $\boldsymbol{D}[k][N \times 2^l]$. The result read out is a two-dimensional matrix $\boldsymbol{A}_k[N \times e_k]$. Matrix $\boldsymbol{A}_k$ represents the match result of the kth stage, and the elements in $\boldsymbol{A}_k$ are 0 or 1. Except for the final stage, the output of the kth stage is a two-dimensional matrix $\boldsymbol{M}_k[N \times 3]$, representing the intermediate match result of the stages that have been processed (stage 0–stage k, where $k = 0, 1, \ldots, s-2$). The physical meanings of the elements in the matrix $\boldsymbol{M}_k$ are shown in Table 3.

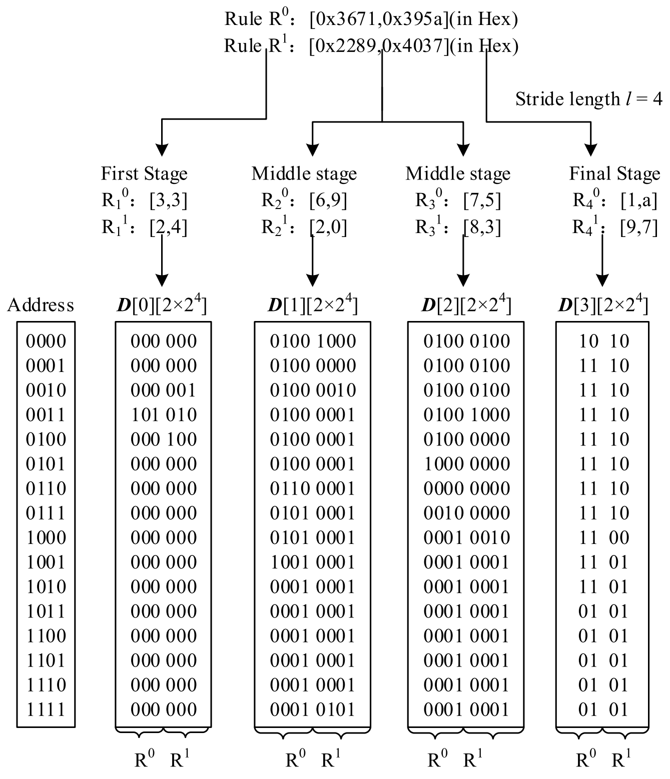
The output of the kth stage $\boldsymbol{M}_k$ is derived from $\boldsymbol{A}_k$ and $\boldsymbol{M}_{k-1}$ by a logic function, i.e.,

$$\boldsymbol{M}_k = G_k(\boldsymbol{A}_k, \boldsymbol{M}_{k-1}), \tag{1}$$

where G_k is the logic function of the kth stage. The details of the logic functions are described in Algorithms 2–4 for different k. The output of the final stage is matrix $\boldsymbol{M}_{s-1}[N \times 1]$. If $M_{s-1}[j] = 1$, it means that $L^j \leq K \leq U^j$, i.e., the input field K matches rule R^j. The lookup process is described by a flowchart in Fig. 5.

Algorithm 1. Rule encoding and storing of RSBV

```
Input: Range matching rule set $\boldsymbol{R}$, bit width $L$, Stride length $l$, and number of rules $N$.
Output: Encoded rule set $\boldsymbol{D}[s \times N \times 2^l]$.
1: Split $\boldsymbol{R}$ into subrule sets $\boldsymbol{R_0}, \boldsymbol{R_1}, \dots, \boldsymbol{R_{s-1}}$ with bit width $l$, where $s = \lceil L/l \rceil$;
2: for $k = 0$ to $s - 1$ do
3:   if $k == 0$ then
4:     /* first stage */
5:     for $j = 0$ to $N - 1$ do
6:       for $i = 0$ to $2^l - 1$ do
7:         if $i == L_k^j == U_k^j$ then $D[k][j][i] = 101$;
8:         else if $i == L_k^j$ then $D[k][j][i] = 001$;
9:         else if $i == U_k^j$ then $D[k][j][i] = 100$;
10:        else if $L_k^j < i < U_k^j$ then $D[k][j][i] = 010$;
11:        else $D[k][j][i] = 000$;
12:        end if
13:      end for
14:    end for
15:  else if $0 < k < s - 1$ then
16:    /* middle stage */
17:    for $j = 0$ to $N - 1$ do
18:      for $i = 0$ to $2^l - 1$ do
19:        if $i == L_k^j == U_k^j$ then $D[k][j][i] = 1010$;
20:        else if $i == L_k^j < U_k^j$ then $D[k][j][i] = 0110$;
21:        else if $i == U_k^j > L_k^j$ then $D[k][j][i] = 1001$;
22:        else if $L_k^j < i < U_k^j$ then $D[k][j][i] = 0101$;
23:        else if $i == L_k^j$ then $D[k][j][i] = 0010$;
24:        else if $i == U_k^j$ then $D[k][j][i] = 1000$;
25:        else if $i < U_k^j$ then $D[k][j][i] = 0100$;
26:        else if $i > L_k^j$ then $D[k][j][i] = 0001$;
27:        else $D[k][j][i] = 000$;
28:        end if
29:      end for
30:    end for
31:  else
32:    /* $k == s - 1$, final stage */
33:    for $j = 0$ to $N - 1$ do
34:      for $i = 0$ to $2^l - 1$ do
35:        if $L_k^j \le i \le U_k^j$ then $D[k][j][i] = 11$;
36:        else if $i \le U_k^j$ then $D[k][j][i] = 10$;
37:        else if $i \ge L_k^j$ then $D[k][j][i] = 01$;
38:        else $D[k][j][i] = 00$;
39:        end if
40:      end for
41:    end for
42:  end if
43: end for
```



Address	$\boldsymbol{D}[0][2\times2^4]$	$\boldsymbol{D}[1][2\times2^4]$	$\boldsymbol{D}[2][2\times2^4]$	$\boldsymbol{D}[3][2\times2^4]$
0000	000 000	0100 1000	0100 0100	10 10
0001	000 000	0100 0000	0100 0100	11 10
0010	000 001	0100 0010	0100 0100	11 10
0011	101 010	0100 0001	0100 1000	11 10
0100	000 100	0100 0001	0100 0000	11 10
0101	000 000	0100 0001	1000 0000	11 10
0110	000 000	0110 0001	0000 0000	11 10
0111	000 000	0101 0001	0010 0000	11 10
1000	000 000	0101 0001	0001 0010	11 00
1001	000 000	1001 0001	0001 0001	11 01
1010	000 000	0001 0001	0001 0001	11 01
1011	000 000	0001 0001	0001 0001	01 01
1100	000 000	0001 0001	0001 0001	01 01
1101	000 000	0001 0001	0001 0001	01 01
1110	000 000	0001 0001	0001 0001	01 01
1111	000 000	0001 0101	0001 0001	01 01

Fig. 4. A simple example of encoding and storage.

An example of the lookup algorithm is shown in Fig. 6. An input field is $K =$ 0x3672 in hex. The first subfield is $K_0 = 3$. Use K_0 as address to read the encoded matrix, then $\boldsymbol{A}_0 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ is returned. According to Algorithm 2, the output of the first stage is $\boldsymbol{M}_0 = \boldsymbol{A}_0 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$. The input of stage 1 is $K_1 = 6$. Read address 6 and we have $\boldsymbol{A}_1 = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. According to Algorithm 3, the output of stage 1 is $\boldsymbol{M}_1 = G_1(\boldsymbol{A}_1, \boldsymbol{M}_0) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$. The input of stage 2 is $K_2 = 7$. Read address 7 and

Table 3. Physical meaning of the output of stage k.

Stage	Output result	Physical meaning
$0 \le k \le s-2$	$M_k[j][0] = 1$	$K_0 \cdots K_k = U_0^j \cdots U_k^j$
	$M_k[j][1] = 1$	$L_0^j \cdots L_k^j < K_0 \cdots K_k < U_0^j \cdots U_k^j$
	$M_k[j][2] = 1$	$K_0 \cdots K_k = L_0^j \cdots L_k^j$
$k = s-1$	$M_k[j][0] = 1$	$L^j \le K \le U^j$

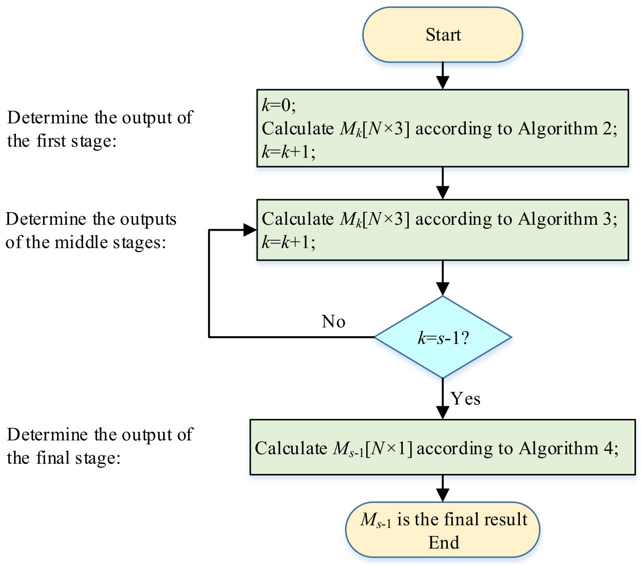
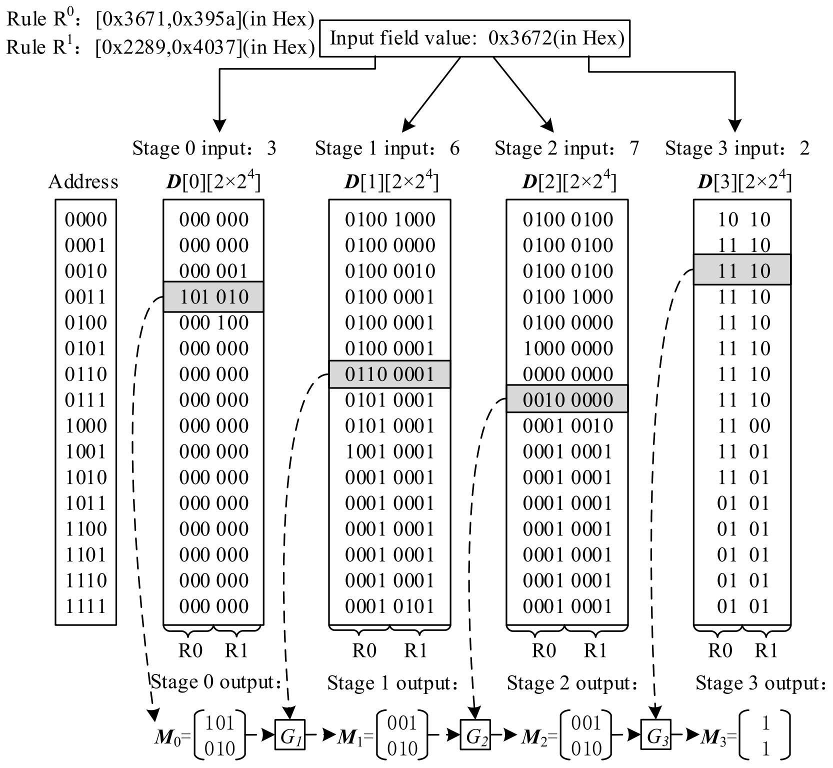

Start
Determine the output of the first stage:
k=0;
Calculate Mk[N×3] according to Algorithm 2;
k=k+1;
Determine the outputs of the middle stages:
Calculate Mk[N×3] according to Algorithm 3;
k=k+1;
No
k=s-1?
Yes
Determine the output of the final stage:
Calculate Ms-1[N×1] according to Algorithm 4;
Ms-1 is the final result
End

Fig. 5. Flowchart of the procedure of RSBV lookup.



Address	D[0][2×2^4]	D[1][2×2^4]	D[2][2×2^4]	D[3][2×2^4]
0000	000 000	0100 1000	0100 0100	10 10
0001	000 000	0100 0000	0100 0100	11 10
0010	000 001	0100 0010	0100 0100	11 10
0011	101 010	0100 0001	0100 1000	11 10
0100	000 100	0100 0001	0100 0000	11 10
0101	000 000	0100 0001	1000 0000	11 10
0110	000 000	0110 0001	0000 0000	11 10
0111	000 000	0101 0001	0010 0000	11 10
1000	000 000	0101 0001	0001 0010	11 00
1001	000 000	1001 0001	0001 0001	11 01
1010	000 000	0001 0001	0001 0001	11 01
1011	000 000	0001 0001	0001 0001	01 01
1100	000 000	0001 0001	0001 0001	01 01
1101	000 000	0001 0001	0001 0001	01 01
1110	000 000	0001 0001	0001 0001	01 01
1111	000 000	0001 0101	0001 0001	01 01

Fig. 6. A simple example of RSBV lookup.

we have $\boldsymbol{A}_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$. According to Algorithm 3, the output of stage 2 is $\boldsymbol{M}_2 = G_2(\boldsymbol{A}_2, \boldsymbol{M}_1) = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$. The input of stage 3 is $K_3 = 2$. Read address 2 and we have $\boldsymbol{A}_3 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$. According to Algorithm 4, the output of stage 3 is $\boldsymbol{M}_3 = G_3(\boldsymbol{A}_3, \boldsymbol{M}_2) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. It is the final result. It means the input field K matches R^0 and R^1. Since rule R^0 has a higher priority, the final result is matching R^0.

Algorithms 2–4 represent the process of range matching for one field. If a rule set contains multiple range matching fields, the algorithms can be easily extended to handle multiple range matching fields. Just repeat the steps of encoding, storage, and lookup for each range matching field.

Algorithm 2. Determine the output of the first stage

```
Input: The input subfield $K_k$, $k = 0$.
Output: Output of the first stage $\boldsymbol{M}_k[N \times 3]$, $k = 0$.
 1: Use $K_k$ as the address to read the encoded subrule set $\boldsymbol{D}[k][N \times 2^l]$; the result read out is
    $\boldsymbol{A}_k[N \times e_k]$;
 2: Initialize $\boldsymbol{M}_k[N \times 3]$ to all zeros;
 3: for $j = 0$ to $N - 1$ do
 4:    for $i = 0$ to $2$ do
 5:       $M_k[j][i] = A_k[j][i]$;
 6:    end for
 7: end for
```

Algorithm 3. Determine the output of the middle stage

```
Input: The output of the previous stage $\boldsymbol{M}_{k-1}[N \times 3]$, input subfield $K_k$, $0 < k < s - 1$.
Output: Output of the middle stage $\boldsymbol{M}_k[N \times 3]$, $0 < k < s - 1$.
 1: Use $K_k$ as the address to read the encoded subrule set $\boldsymbol{D}[k][N \times 2^l]$; the result read out is
    $\boldsymbol{A}_k[N \times e_k]$;
 2: Initialize $\boldsymbol{M}_k[N \times 3]$ to all zeros;
 3: for $j = 0$ to $N - 1$ do
 4:    if $\boldsymbol{M}_{k-1}[j] == **1$ and $\boldsymbol{A}_k[j] == **1*$ then
 5:       $M_k[j][2] = 1$;
 6:    end if
 7:    if $(\boldsymbol{M}_{k-1}[j] == 100$ and $\boldsymbol{A}_k[j] == *1**)$ or $(\boldsymbol{M}_{k-1}[j] == 001$ and $\boldsymbol{A}_k[j] == ***1)$ or
       $(\boldsymbol{M}_{k-1}[j] == 010)$ or $(\boldsymbol{M}_{k-1}[j] == 101$ and $\boldsymbol{A}_k[j] == *1*1)$ then
 8:       $M_k[j][1] = 1$;
 9:    end if
10:    if $\boldsymbol{M}_{k-1}[j] == 1**$ and $\boldsymbol{A}_k[j] == 1***$ then
11:       $M_k[j][0] = 1$;
12:    end if
13: end for
```

Algorithm 4. Determine the output of the final stage

```
Input: The output of the previous stage $\boldsymbol{M}_{k-1}[N \times 3]$, input subfield $K_k$, $k = s - 1$.
Output: Output of the final stage $\boldsymbol{M}_k[N \times 1]$, $k = s - 1$.
1: Use $K_k$ as the address to read the encoded subrule set $\boldsymbol{D}[k][N \times 2^l]$; the result read out is $\boldsymbol{A}_k[N \times e_k]$;
2: Initialize $\boldsymbol{M}_k[N \times 1]$ to all zeros;
3: for $j = 0$ to $N - 1$ do
4:     if $(\boldsymbol{M}_{k-1}[j] == 100$ and $\boldsymbol{A}_k[j] == 1*)$ or $(\boldsymbol{M}_{k-1}[j] == 001$ and $\boldsymbol{A}_k[j] == *1)$ or $(\boldsymbol{M}_{k-1}[j] == 010)$ or $(\boldsymbol{M}_{k-1}[j] == 101$ and $\boldsymbol{A}_k[j] == 11)$ then
5:         $M_k[j][0] = 1$;
6:     end if
7: end for
```

2.3. *FPGA implementation of RSBV*

2.3.1. *Implementation details*

The FPGA implementation of RSBV is a two-dimensional pipelined architecture as shown in Fig. 7. This architecture consists of multiple modular processing elements. In the horizontal direction, the rule set is split into multiple subfields with bit width of l. In the vertical direction, the rule set is split into multiple subrule blocks with size of n. There are totally $X = F \times \lceil L/l \rceil$ subfields, and $Y = \lceil N/n \rceil$ subrule blocks. The

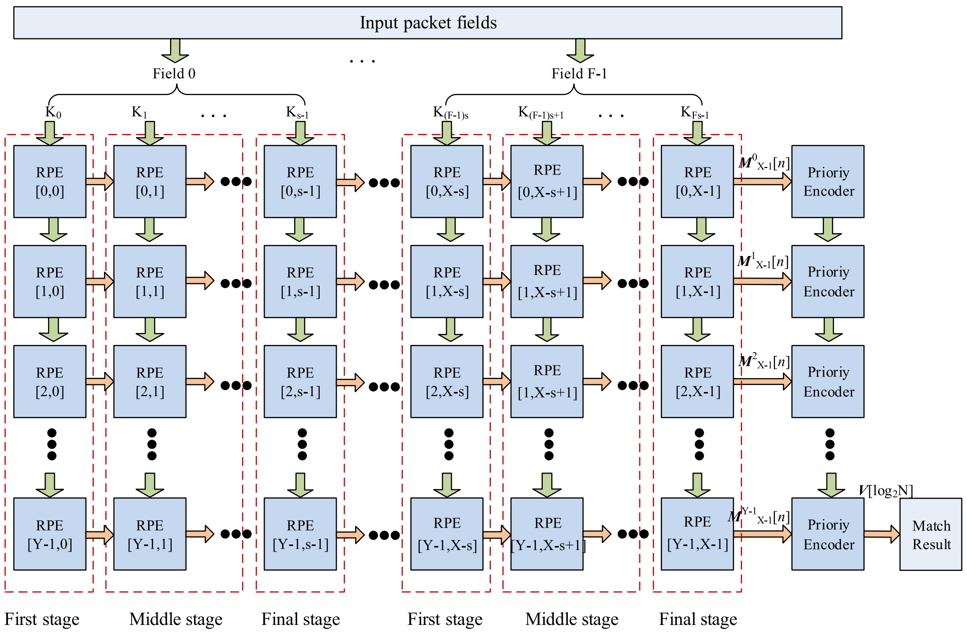


Fig. 7. The architecture of the FPGA implementation of RSBV.

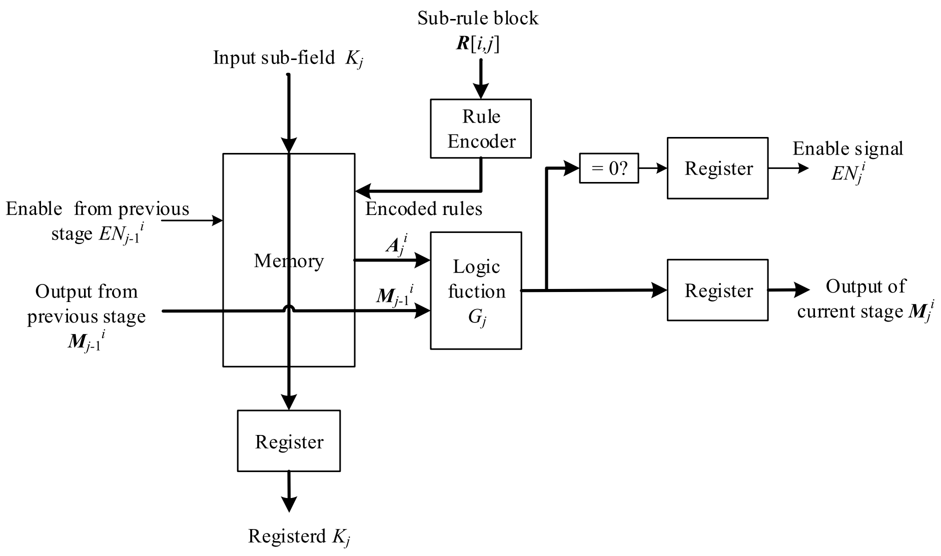


Fig. 8. Internal structure of RPE$[i, j]$.

module for processing an l-bit subfield against n subrules is called a range processing element (RPE). The pipeline operation is performed simultaneously in the horizontal direction and the vertical direction. This architecture ensures that the range matching of an input packet header can be completed every clock cycle, and line-rate processing is guaranteed.

Let RPE$[i, j]$ denote the RPE located in ith row and jth column. The internal structure of RPE$[i, j]$ is shown in Fig. 8. Its function is described as follows. It first receives the subrule block $\boldsymbol{R}[i, j]$, then encodes the rules and stores them in its local memory. During a lookup, the input signals of RPE$[i, j]$ are input subfield K_j, enable signal from the previous stage EN_{j-1}^i, and the output of the previous stage $\boldsymbol{M}_{j-1}^i$. If $\mathrm{EN}_{j-1}^i \neq 0$, then K_j is used as an address to read the memory and $\boldsymbol{A}_j^i$ is obtained. The output of RPE$[i, j]$ is the signal $\boldsymbol{M}_j^i$, and $\boldsymbol{M}_j^i$ is calculated by

$$\boldsymbol{M}_j^i = G_j(\boldsymbol{A}_j^i, \boldsymbol{M}_{j-1}^i)\,, \tag{2}$$

where G_j is the logic function of the jth stage as described in Algorithms 2–4 for different j. Finally $\boldsymbol{M}_j^i$ is registered and output. If $\boldsymbol{M}_j^i$ is all-zero, then $\mathrm{EN}_j^i = 0$, else $\mathrm{EN}_j^i = 1$. The EN signals are used for power gating. If the current RPE outputs all-zero, it means that this packet header does not match any of the n rules; so that the subsequent stages are deactivated to save the power because there is no need to match the remaining subfields for this packet.

Algorithm 5. The operation of the priority encoder in the ith row

```
Input: The output of the previous RPE $\boldsymbol{M}^i_{X-1}[n \times 1]$, output of the previous priority encoder $\boldsymbol{V}^{i-1}[\log_2 N]$.
Output: Output of the priority encoder in the $i$th row $\boldsymbol{V}^i[\log_2 N]$.
 1: Define binary string $\boldsymbol{T}[\log_2 N]$ and initialize it to all zeros;
 2: for $k = 0$ to $n - 1$ do
 3:     if $M^i_{X-1}[k] == 1$ then
 4:         $\boldsymbol{T}[\log_2 N] = toBinaryString(k + i \times n)$;
 5:         /* Convert integer $k + i \times n$ to binary string to represent the rule number. */
 6:         break;
 7:     end if
 8: end for
 9: if strcmp($\boldsymbol{T}[\log_2 N], \boldsymbol{V}^{i-1}[\log_2 N]$) $< 0$ then
10:     $\boldsymbol{V}^i[\log_2 N] = \boldsymbol{T}[\log_2 N]$;
11: else
12:     $\boldsymbol{V}^i[\log_2 N] = \boldsymbol{V}^{i-1}[\log_2 N]$;
13: end if
```

In the above pipeline architecture, the last module of each row is the priority encoder. The function of the priority encoder in the ith row is to determine the rule with the highest priority among all the matched rules, and output the rule number in binary format. The operation of priority encoder in the ith row is described in Algorithm 5. The overall processing flow of the two-dimensional pipelined architecture is described in Algorithm 6.

2.3.2. *Performance analysis of RSBV*

(1) Space complexity

First of all, the space complexity of RSBV is analyzed for a rule set $\boldsymbol{R}$ with bit width L, rule number N, and field number F. At the first stage, each rule requires a 3-bit encoding. The memory depth of each stage is 2^l. Therefore, the memory space cost by the first stage is

$$S_{\text{first\_stage}}(\boldsymbol{R}) = 3 \times N \times F \times 2^l\,. \tag{3}$$

Similarly, at the middle stages, each rule requires a 4-bit encoding and the final stage requires 2-bit encoding. There are totally $\lceil L/l \rceil - 2$ middle stages for each field. Thus, the memory space costs by the middle stage and the final stage are obtained as follows:

$$S_{\text{middle\_stage}}(\boldsymbol{R}) = 4 \times (\lceil L/l \rceil - 2) \times N \times F \times 2^l\,, \tag{4}$$

$$S_{\text{final\_stage}}(\boldsymbol{R}) = 2 \times N \times F \times 2^l\,. \tag{5}$$

Algorithm 6. The overall processing flow of the two-dimensional pipelined architecture

```
Input: Range matching rule set $\boldsymbol{R}$, input packet fields $K_0K_1\cdots K_{Fs-1}$.
Output: Final result $\boldsymbol{V}[\log_2 N]$.
 1: Let $X = F \times \lceil L/l \rceil$, $Y = \lceil N/n \rceil$;
 2: Call Algorithm 1 to encode the rule set $\boldsymbol{R}$ to three-dimensional matrix $\boldsymbol{D}[X \times N \times 2^l]$;
 3: for $i = 0$ to $Y - 1$ do
 4:    for $j = 0$ to $X - 1$ do
 5:       RPE$[i, j]$.memory $= \boldsymbol{D}[j][i \times n \sim (i+1) \times n - 1][0 \sim 2^l - 1]$;
 6:    end for
 7: end for
 8: for $i = 0$ to $Y - 1$ do
 9:    for $j = 0$ to $X - 1$ do
10:       if $\mathrm{EN}^i_{j-1} == 0$ then
11:          $\mathrm{EN}^i_j = 0$, $\boldsymbol{M}^i_j = \boldsymbol{0}$;
12:       else
13:          if $j\%s == 0$ then
14:             Call Algorithm 2 to obtain $\boldsymbol{M}^i_j$; /* First stage RPE */
15:          else if $(j+1)\%s == 0$ then
16:             Call Algorithm 4 to obtain $\boldsymbol{M}^i_j$; /* Final stage RPE */
17:          else
18:             Call Algorithm 3 to obtain $\boldsymbol{M}^i_j$; /* Middle stage RPE */
19:          end if
20:          if $\boldsymbol{M}^i_j == \boldsymbol{0}$ then
21:             $\mathrm{EN}^i_j = 0$;
22:          else
23:             $\mathrm{EN}^i_j = 1$;
24:          end if
25:       end if
26:       if $j == X - 1$ then
27:          Call Algorithm 5 to obtain the rule number of $i$th row $\boldsymbol{V}^i[\log_2 N]$;
28:       end if
29:    end for
30:    if $i == Y - 1$ then
31:       The final result is $\boldsymbol{V}[\log_2 N] = \boldsymbol{V}^{Y-1}[\log_2 N]$, which represents the matched rule number;
32:    end if
33: end for
```

From Eqs. (3)–(5), the theory space complexity of RSBV is given by the following equation:

$$\begin{aligned} S_{\text{Theory}}(\boldsymbol{R}) &= S_{\text{first\_stage}}(\boldsymbol{R}) + S_{\text{middle\_stage}}(\boldsymbol{R}) + S_{\text{final\_stage}}(\boldsymbol{R}) \\ &= \left(3 + 4 \times \left(\left\lceil \frac{L}{l} \right\rceil - 2\right) + 2\right) \times N \times F \times 2^l \\ &= \left(4 \times \left\lceil \frac{L}{l} \right\rceil - 3\right) \times N \times F \times 2^l . \end{aligned} \tag{6}$$

It can be found that the space complexity increases exponentially with the increment of Stride length l. However, the selection of l needs to take the characteristic of the internal memory modules of FPGA into consideration, because that different RPEs need to be operated in parallel. But in practical devices, the granularity of memory resources that can be operated in parallel is limited. In Xilinx FPGA series,[33] the block RAM (BRAM) has two specifications of 18-Kb and 36-Kb. The minimum granularity of the 18-Kb primitive is 36 bits $\times$ 512. Similarly, the minimum granularity of the 36-Kb primitive is 72 bits $\times$ 512. Therefore, the minimum depth of BRAM that can be operated in parallel is 512. If l is too small, a large amount of the memory space will be wasted. Let the required memory depth in each RPE be d, then we have

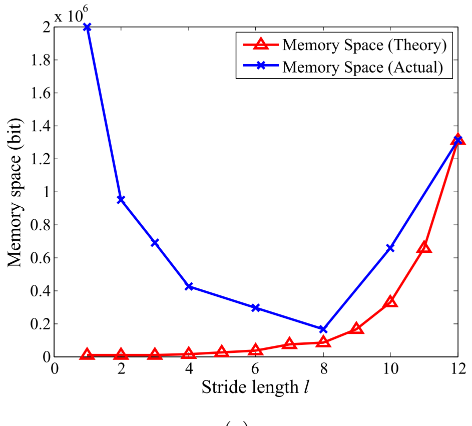
$$d = \begin{cases} 512, & 2^l \le 512, \\ \left\lceil \dfrac{2^l}{512} \right\rceil \times 512, & 2^l > 512\,. \end{cases} \tag{7}$$

Then we obtain the actual space complexity of RSBV as follows:

$$S_{\text{Actual}} = \left(4 \times \left\lceil \frac{L}{l} \right\rceil - 3\right) \times N \times F \times d\,. \tag{8}$$

When $L = 16$, $N = 64$, and $F = 1$, the space complexity as a function of l is shown in Fig. 9(a). It is found that the optimal Stride length l is 8 in the FPGA implementation.

(2) Time complexity



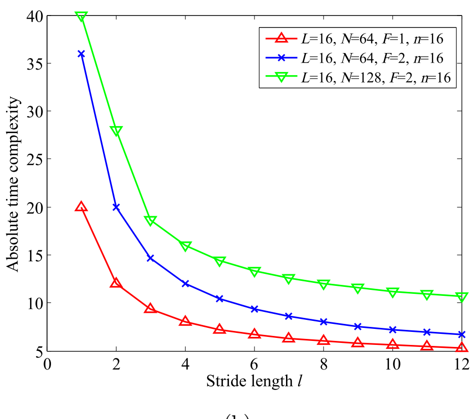


Fig. 9. Simulation of space complexity and time complexity for RSBV. (a) Space complexity of RSBV versus Stride length ($L = 16$, $N = 64$, $F = 1$) and (b) Absolute time complexity of RSBV versus Stride length.

The time complexity of RSBV depends on the number of RPEs. Let the columns and rows of the two-dimensional RPE array be X and Y, respectively, then we have

$$X = \left\lceil \frac{L}{l} \right\rceil \times F\,, \tag{9}$$

$$Y = \left\lceil \frac{N}{n} \right\rceil. \tag{10}$$

Then we obtain the absolute time complexity of RSBV as follows:

$$\begin{aligned} T_{\text{Absolute}}(\text{RSBV}) &= O(X + Y + 1) \\ &= O\left(\left\lceil \frac{L}{l} \right\rceil \times F + \left\lceil \frac{N}{n} \right\rceil\right), \end{aligned} \tag{11}$$

where the $O(\cdot)$ notation means the asymptotic upper bound of a given function within a constant factor. Figure 9 provides a simulation on time complexity T_{Absolute} versus Stride length l. It can be found that for all given F and N, the absolute time complexity is inversely proportional to l. The reason is that with the increasing l, the number of stages required for the pipeline decreases, leading to a lower value of the time complexity. However, the RSBV architecture enables highly parallel two-dimensional pipeline operations, which has a relative time complexity of $O(1)$. Here, the absolute time complexity represents the delay between the input packet and the output of matching result. The relative time complexity is the minimum interval between two matching results. Therefore, the processing speed of RSBV is independent of rule size, and the line-rate processing can be achieved.

3. Flow Classification for any Types of Fields: AFBV

The RSBV algorithm described in the previous section can process range matching in a very high speed, but it cannot handle exact matching and wildcard matching because the encoding schemes are different. However, if the entire rule set is first divided into range matching fields and other fields, the different fields can be processed by different methods in parallel. By this way, line-rate processing of any type of field can be achieved. In this section, we introduce the AFBV method. The key idea of AFBV is to split the fields into two types: range matching fields and other fields. Other fields include the fields of exact matching, prefix matching, and arbitrary wildcard matching. The range matching fields are processed by RSBV algorithm, while other fields are processed by two-dimensional pipelined Stride BV algorithm.[31] The architecture of AFBV is shown in Fig. 10.

We use the 13 required fields at the ingress port in OpenFlow 1.5 as an example to illustrate the workflow of AFBV algorithm. First, the fields in the rule set are split

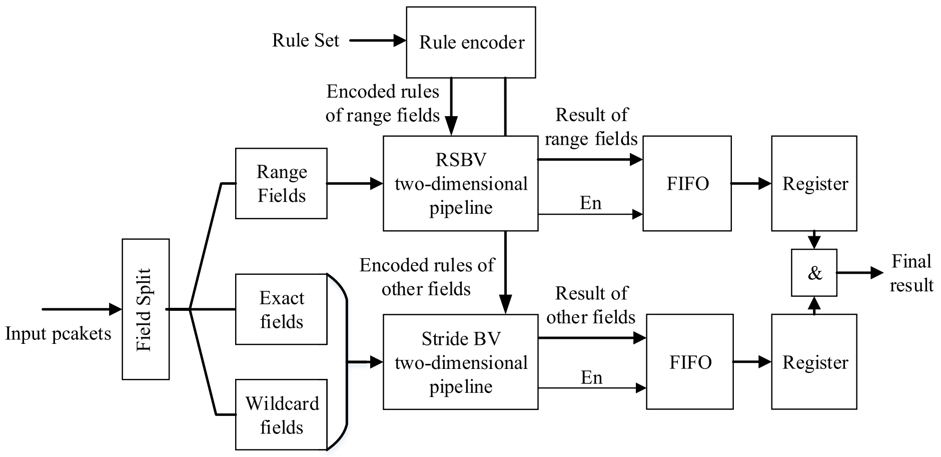


Fig. 10. Architecture of AFBV.

into two types. The range fields include {TCP_SRC, TCP_DST, UDP_SRC, UDP_DST}. The other fields include {IN_PORT, ETH_SRC, ETH_DST, ETH_TYPE, IP_PROTO, IPv4_SRC, IPv4_DST, IPv6_SRC, IPv6_DST}. The RSBV algorithm described in Sec. 2 is used to process the range fields. While the other fields are processed by two-dimensional pipelined Stride BV simultaneously. Due to the fact that the absolute delays of Stride BV and RSBV are different, the results need to be stored in FIFOs after output. When the algorithm which has a longer delay finishes, the results are read from the FIFOs to obtain the final result by a logic “AND”operation.

The absolute time complexity of AFBV is

$$\begin{aligned} T_{\text{Absolute}}(\text{AFBV}) &= O(\max\{T_{\text{Absolute}}(\text{Stride BV}), T_{\text{Absolute}}(\text{RSBV})\} + 2) \\ &= O\left(\max\left\{\left\lceil\frac{L_{\text{Range}}}{l}\right\rceil \times F_{\text{Range}}, \left\lceil\frac{L_{\text{Other}}}{l}\right\rceil \times F_{\text{Other}}\right\} + \left\lceil\frac{N}{n}\right\rceil\right) \\ &= O\left(\left\lceil\frac{L_{\text{Total}}}{l}\right\rceil \times F + \left\lceil\frac{N}{n}\right\rceil\right). \end{aligned} \tag{12}$$

However, the relative time complexity of AFBV is $O(1)$, since the relative time complexities of both two-dimensional Stride BV and RSBV are $O(1)$.

4. Performance Simulation

In this section, the performance of AFBV is evaluated in FPGA platform. The experimental environment includes Xilinx Virtex 7 xc7vx690t FPGA device, Vivado 2016.2, and ModelSim 10.5. We generate rule sets with different sizes based on the 13 required fields defined in OpenFlow 1.5. The total bit width is 512 bits, including four

Table 4. Throughputs comparison.

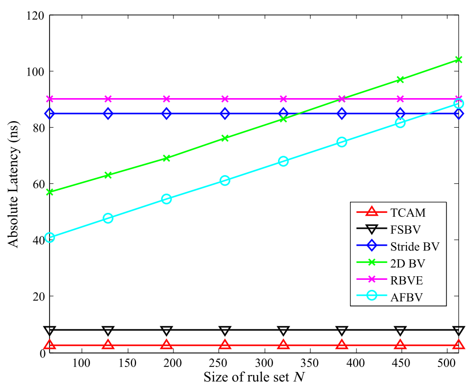
Algorithm	Throughput	Space complexity	Range supported
TCAM[16]	400 MPPS	$O([2(L-1)]^F \times N)$	yes
FSBV[28]	360 MPPS	$O(L \times N \times F)$	no
Stride BV[29]	400 MPPS	$O((L/l) \times N \times F \times 2^l)$	no
2D BV[31]	650 MPPS	$O((L/l) \times N \times F \times 2^l)$	no
RBVE[32]	380 MPPS	$O((L/l) \times N \times F \times 2^l)$	yes
AFBV	520 MPPS	$O((L/l) \times N \times F \times 2^l)$	yes

range matching fields (64 bits). The performance indicators include throughput, latency, FPGA resource utilization, and power consumption.

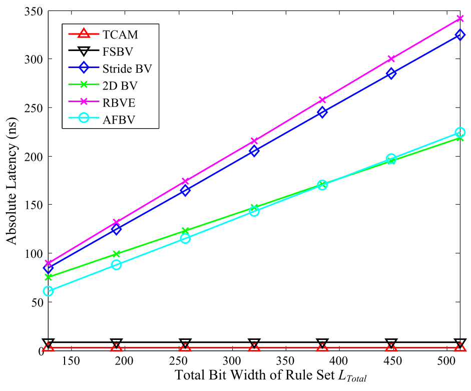
4.1. *Throughput*

In the first experiment, the throughputs of multiple flow classification are compared. When the parameters are set as $L_{\text{Total}} = 512$, $N = 1,024$, $l = 8$, and $n = 36$, the simulation result shows that the maximum clock frequency of AFBV algorithm is up to 260 MHz. If both block RAM and distributed RAM used in the AFBV algorithm are set to true dual-port RAM, the throughput of the algorithm can be doubled to 520 MPPS. To our knowledge, it is the first range supported approach that can achieve a throughput of 520 MPPS. A comparison of the AFBV algorithm and existing works is shown in Table 4. It can be found that the proposed algorithm achieves a highest throughput while effectively supporting the range matching. The throughput is improved by 44% compared with FSBV, 30% compared with Stride BV, and 37% compared with RBVE. The main reason is that in these benchmarking methods, the number of memory modules required for each PE grows linearly with N. This means the length of the longest wire connecting different memory modules in a PE also increases at $O(N)$ rate, which degrades the throughput of the pipeline for large N. But in the proposed method, a two-dimensional pipelined architecture is used to address this problem. The two-dimensional pipeline is constructed by multiple modular RPEs, and each RPE matches l-bit header field against n rules where n is a constant. Therefore, the widths of the internal signals are reduced to constant level and the clock frequency is improved significantly.

The throughput of the proposed AFBV algorithm is 20% lower than the two-dimensional Stride BV. The main reason is that the processing of range fields requires more complex logic operations and more memory resources, resulting in a certain decline of the clock frequency. However, 2D BV cannot support range matching. Besides, the space complexity is linearly related to the number of rules N and the number of fields F. The problem of prefix expansion is eliminated, so that it is suitable for processing multi-field flow classification rule set. The throughputs of different flow classification methods are further shown in Fig. 12(a).



(a)

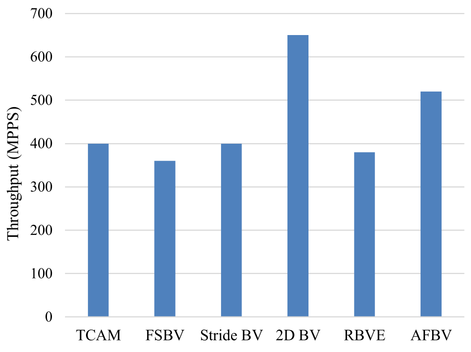


(b)

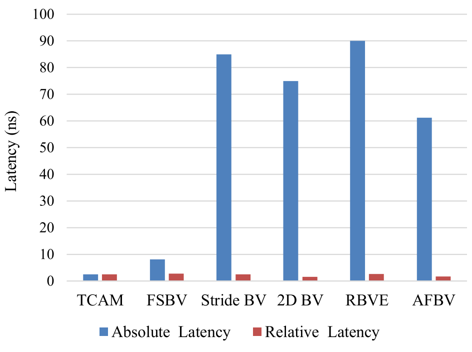
Fig. 11. Absolute latencies comparison. (a) Absolute latency versus rule set size N when $L_{\text{Total}} = 128$ and (b) Absolute latency versus total bit width L_{Total} when $N = 256$.

4.2. *Latency*

In the second experiment, the latency performance with respect to various values of N and L is analyzed. First, the absolute latencies of multiple flow classification methods versus the rule set size N when $L_{\text{Total}} = 128$ are shown in Fig. 11(a). It can be seen that the absolute latencies of 2D BV and AFBV increase linearly with N. This is because that 2D BV and AFBV utilize two-dimensional pipeline architecture. The greater the number of rules, the greater is the number of stages in the vertical direction, so that there is a higher pipeline latency. The absolute latency of AFBV is smaller than 2D BV, because AFBV matches range fields and other fields in parallel while 2D BV is sequential. The latencies of FSBV, Stride BV, and RBVE are independent of N, because these methods do not use the pipelined structure in



(a)



(b)

Fig. 12. Throughputs and latencies comparison. (a) Throughput comparison for different flow classification methods and (b) Absolute latency and relative latency of multiple flow classification methods ($N = 256$, $L_{\text{Total}} = 128$).

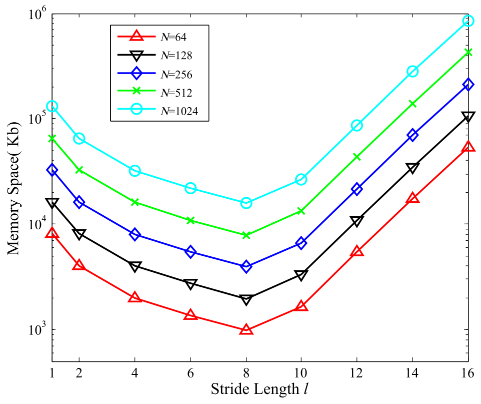
the vertical direction. All N rules are matched at the same time in these methods. The TCAM has the lowest absolute latency, which can achieve single-cycle classification.

Second, the absolute latencies versus total bit width L_{Total} when $N = 256$ are shown in Fig. 11(b). It can be found that on one hand, the absolute latencies of Stride BV, 2D BV, RBVE, and AFBV increase linearly with L_{Total}. With the increment of L_{Total}, the number of stages in the horizontal direction increases, leading to a higher pipeline latency. The RBVE has the highest absolute latency due to the lowest clock frequency. On the other hand, the absolute latencies of 2D BV and AFBV are quite close because they both utilize the two-dimensional pipeline architecture. In addition, TCAM's absolute latency is still the smallest.

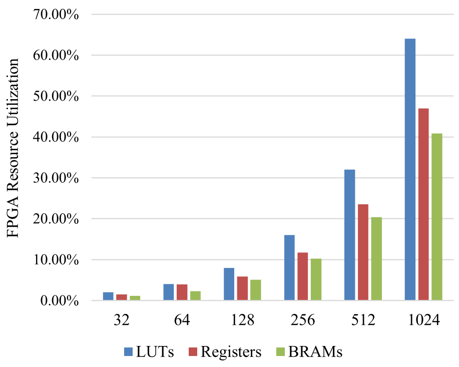
Although the absolute latency is higher than TCAM and FSBV, the relative latency of AFBV is one clock cycle. The comparison of absolute latency and relative latency of multiple flow classification methods when $N = 256$ and $L_{\text{Total}} = 128$ is shown in Fig. 12(b). It can be seen that the absolute latency of AFBV algorithm is the smallest. The main reason is that, first, the AFBV eliminates long internal signals and increases the clock rate due to the deeply pipelined architecture. Second, AFBV processes range matching, exact matching, and wildcard matching in parallel, rather than in sequential, so that the stages of the pipeline in the horizontal direction are reduced, leading to a lower pipeline latency.

4.3. *FPGA resource utilization*

In the third experiment, we focus on the FPGA resource utilization of the proposed AFBV algorithm. First, the memory resource cost of AFBV as a function of l for different N is shown in Fig. 13(a). It is found that the optimal Stride length is $l = 8$, which agrees with the theoretical analysis well. Next, FPGA resource utilization as a



(a)



(b)

Fig. 13. FPGA resource cost and utilization of AFBV. (a) Memory resource cost as a function of l for different N and (b) FPGA resource utilization as a function of N when $L_{\text{Total}} = 512$.

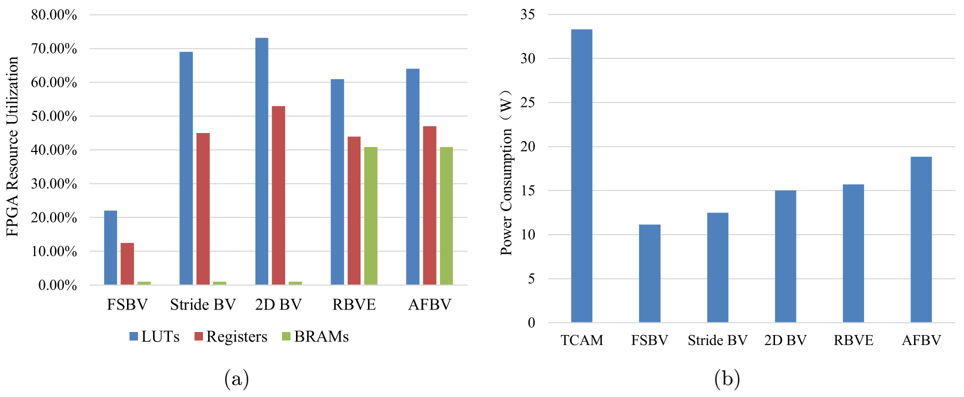


Fig. 14. FPGA resource utilizations and power consumptions comparison. (a) FPGA resource utilization for different flow classification methods and (b) Power consumption for different flow classification methods.

function of N when $L_{\text{Total}} = 512$, $l = 8$, and $n = 36$ is shown in Fig. 13(b). As can be seen, the resource utilization on FPGA scales linearly with N. The largest design supported on FPGA depends on the amount of available resources. The largest design in the experiment is a 13-field rule set (512 bits, including four range matching fields of 64 bits) with 1 k rules. It is the largest to our knowledge.

For the rule set above, a comparison of FPGA resource utilization for different flow classification methods is shown in Fig. 14(a). We can see that FSBV has the lowest resource cost. However, its throughput is also the lowest because it does not employ pipelined architecture, leading to long internal signals and a reduced clock frequency. Both Stride BV and 2D BV use on-chip distRAM as memory, which will consume a large amount of logic resources. AFBV and RBVE use FPGA's on-chip BRAM to store range matching rules, and as a result, a certain amount of logic resources can be saved. In general, a large number of BRAM resources are often built in FPGA chips (52-Mb BRAM resources built in Virtex 7 xc7vx690t FPGA). Therefore, the additional BRAM cost will not become a bottleneck, but it effectively solves the problem of range matching.

4.4. *Power consumption*

Finally, the power consumptions for different flow classification methods are analyzed. For the rule set above, a comparison is shown in Fig. 14(b). The TCAM power is calculated using an 18-Mb TCAM,[15] and the power consumed by the extra logic for TCAM control is ignored. The powers of other FPGA-based methods are obtained by the Vivado development tool. It can be seen that the power of AFBV is higher than Stride BV and 2D BV because AFBV uses BRAMs to store the range matching rules, and BRAMs will consume a relatively larger power (about 0.185 W/Mb at 260 MHz) than logic resources in FPGA. But compared with TCAM, AFBV algorithm can reduce the power consumption by about 43%. This is because the TCAM is very power hungry due to the completely parallel searching. All addresses in TCAM are activated when a packet is being matched. The proposed method uses power gating to deactivate the subsequent stages if an early stage reports no match, so that a significant amount of power can be saved on average.

5. Conclusion

Network flow classification is a key technology in high-performance switches and routers. Currently, the increasing bandwidth requirement and more complex rules make the flow classification more and more challenging. The software-based classification method cannot meet the performance requirement as high as 100 Gbps, and the FPGA-based classification lacks effective support for range matching. For this, this paper proposes the RSBV method which supports range matching while guaranteeing a high throughput. Based on RSBV, a flow classification method for any types of fields called AFBV is proposed, which can support exact matching, longest prefix matching, range matching, and arbitrary wildcard matching simultaneously. The proposed methods are implemented and evaluated on FPGA platform. The experimental results show that the AFBV can achieve a throughput of 520 MPPS and supports the flow classification for multi-dimensional fields including range matching.

Acknowledgments

This work was supported in part by the Project of Science and Technology on Information Transmission and Dissemination in Communication Networks Laboratory (KX152600010/ITD-U15001), the National Natural Science Foundation of China (Grant Nos. 61306047 and 61502204), the Fundamental Research Funds for the Central Universities (JB140112), and the Qing Lan Project of Jiangsu.

References

1. A. Fiessler, C. Lorenz, S. Hager, B. Scheuermann and A. W. Moore, HyPaFilter+: Enhanced hybrid packet filtering using hardware assisted classification and header space analysis, *IEEE/ACM Trans. Netw.* **25** (2017) 3655–3669.
2. W. Pak and Y. J. Choi, High performance and high scalable packet classification algorithm for network security systems, *IEEE Trans. Dependable Secur. Comput.* **14** (2017) 37–49.
3. K. Cao, G. Xu, J. Zhou, T. Wei, M. Chen and S. Hu, QoS-adaptive approximate real-time computation for mobility-aware IoT lifetime optimization, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* (2018), doi: 10.1109/TCAD.2018.2873239.
4. C. Charopoulos, F. Andritsopoulos, Y. Mitsos, G. Doumenis and G. Stasinopoulos, Packet indexing process optimized for high-speed network processors, *J. Circuits Syst. Comput.* **14** (2005) 841–860.

5. S. C. Kao, D. Y. Lee, T. S. Chen and A. Y. Wu, Dynamically updatable ternary segmented aging bloom filter for OpenFlow-compliant low-power packet processing, *IEEE/ACM Trans. Netw.* **26** (2018) 1004–1017.
6. S. Li, D. Hu, W. Fang, S. Ma, C. Chen, H. Huang and Z. Zhu, Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability, *IEEE Netw.* **31** (2017) 58–66.
7. P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese and D. Walker, P4: Programming protocol-independent packet processors, *ACM SIGCOMM Comput. Commun. Rev.* **44** (2014) 87–95.
8. D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmoky and S. Uhlig, Software-defined networking: A comprehensive survey, *Proc. IEEE* **103** (2015) 10–13.
9. D. B. Rawat and S. R. Reddy, Software defined networking architecture, security and energy efficiency: A survey, *IEEE Commun. Surv. Tutor.* **19** (2017) 325–346.
10. Huawei Technologies Co., Ltd., Huawei CE12800 series datacenter switches (2018), http://e.huawei.com/cn/products/enterprise-networking/switches/data-center-switches/ce12800.
11. D. Y. Chang and P. C. Wang, TCAM-based multi-match packet classification using multidimensional rule layering, *IEEE/ACM Trans. Netw.* **24** (2016) 1125–1138.
12. E. Norige, A. X. Liu and E. Torng, A ternary unification framework for optimizing TCAM-based packet classification systems, *IEEE/ACM Trans. Netw.* **26** (2018) 657–670.
13. Y. C. Cheng and P. C. Wang, Scalable multi-match packet classification using TCAM and SRAM, *IEEE Trans. Comput.* **65** (2016) 2257–2269.
14. S. I. Ali, Md. S. Islam and M. R. Islam, A comprehensive review of energy efficient content addressable memory circuits for network applications, *J. Circuits Syst. Comput.* **25** (2016) 1630002.
15. C. R. Meiners, A. X. Liu, E. Torng and J. Patel, Split: Optimizing space, power, and throughput for TCAM-based classification, *Proc. ACM/IEEE Seventh Symp. Architectures for Networking & Communications Systems* (2011), pp. 200–210.
16. K. Lakshminarayanan, A. Rangarajan and S. Venkatachary, Algorithms for advanced packet classification with ternary CAMs, *ACM SIGCOMM Comput. Commun. Rev.* **35** (2005) 193–204.
17. Y. C. Cheng and P. C. Wang, Packet classification using dynamically generated decision Trees, *IEEE Trans. Comput.* **64** (2015) 582–586.
18. O. Erdem and C. F. Bazlamacci, Array design for trie-based IP lookup, *IEEE Commun. Lett.* **14** (2010) 773–775.
19. Y. Qi, L. Xu, B. Yang, Y. Xue and J. Li, Packet classification algorithms: From theory to practice, *Proc. IEEE Conf. Computer Communications (INFOCOM)* (2009), pp. 648–656.
20. M. Kim, L. Liu and W. Choi, A GPU-aware parallel index for processing high-dimensional big data, *IEEE Trans. Comput.* **67** (2018) 1388–1402.
21. W. Jiang and V. K. Prasanna, Data structure optimization for power-efficient IP lookup architectures, *IEEE Trans. Comput.* **62** (2013) 2169–2182.
22. L. Luo, G. Xie, Y. Xie, L. Mathy and K. Salamatian, A hybrid hardware architecture for high-speed IP lookups and fast route updates, *IEEE/ACM Trans. Netw.* **22** (2014) 957–969.
23. J. H. Wee and W. Pak, Fast packet classification based on hybrid cutting, *IEEE Commun. Lett.* **21** (2017) 1011–1014.

24. P. Gupta and N. McKeown, Classifying packets with hierarchical intelligent cuttings, *IEEE Micro* **20** (2000) 34–41.
25. S. Singh, F. Baboescu, G. Varghese and J. Wang, Packet classification using multidimensional cutting, *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications* (2003), pp. 213–224.
26. B. Vamanan, G. Voskuilen and T. N. Vijaykumar, EffiCuts: Optimizing packet classification for memory and throughput, *ACM SIGCOMM Comput. Commun. Rev.* **40** (2010) 207–218.
27. M. Varvello, R. Laufer, F. Zhang and T. V. Lakshman, Multilayer packet classification with graphics processing units, *IEEE/ACM Trans. Netw.* **24** (2016) 2728–2741.
28. W. Jiang and V. K. Prasanna, Field-split parallel architecture for high performance multi-match packet classification using FPGAs, *Proc. ACM Symp. Parallelism in Algorithms and Architectures* (2009), pp. 188–196.
29. T. Ganegedara and V. K. Prasanna, StrideBV: Single chip 400G+ packet classification, *Proc. IEEE Int. Conf. High Performance Switching and Routing (HPSR)* (2012), pp. 1–6.
30. T. Ganegedara, W. Jiang and V. K. Prasanna, A scalable and modular architecture for high-performance packet classification, *IEEE Trans. Parallel Distrib. Syst.* **25** (2014) 1135–1144.
31. Y. R. Qu and V. K. Prasanna, High-performance and dynamically updatable packet classification engine on FPGA, *IEEE Trans. Parallel Distrib. Syst.* **27** (2016) 197–209.
32. Y. K. Chang and C. S. Hsueh, Range enhanced packet classification design on FPGA, *IEEE Trans. Emerg. Top. Comput.* **4** (2016) 214–224.
33. Xilinx, Inc., Xilinx 7 Series FPGAs Configuration User Guide (2018), https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf.