IEEE TRANSACTIONS ON
EMERGING TOPICS
IN COMPUTING

Received 1 December 2014; revised 9 April 2015; accepted 21 May 2015.
Date of Publication 29 June 2015; date of current version 8 June 2016.

Digital Object Identifier 10.1109/TETC.2015.2449666

Range-Enhanced Packet Classification Design on FPGA

YEIM-KUAN CHANG AND CHUN-SHENG HSUEH

Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 701, Taiwan
CORRESPONDING AUTHOR: Y.-K. CHANG (ykchang@mail.ncku.edu.tw)

ABSTRACT The future of fast Internet needs powerful routers to support abundant network functionalities, such as firewall, QoS, and virtual private networks, by classifying the packets into different categories based on a set of predefined rules, so-called multi-field packet classification. Traditional packet classification that considers only 5-tuple fields is not sufficient for today's complicated network requirements. OpenFlow switch was born to take care of these complex requirements using a rule set with the rich definition as the software–hardware interface. This paper considers OpenFlow1.0, consisting of 12-tuple header fields. We propose two schemes to process the range fields. The first scheme has the same characteristic as StrideBV [15] using specially designed codes to store the pre-computed results in memory. The second scheme uses a simple sub-range comparison method to find the matching result in a sequential fashion. To show the performance and compare with other proposed schemes, we implement the proposed schemes on Xilinx Virtex-6 XC6VLX760 FPGA device. Experimental results show that our designs can handle 5 K more OpenFlow rules on Virtex-6 XC6VLX760. To the best of our knowledge, our proposed scheme is the first range supported method that can sustain the throughputs of more than 380 MHz.

INDEX TERMS Packet classification, pipelined architecture, FPGA, OpenFlow.

I. INTRODUCTION

Network security has become more and more important these days because of the various attacks on the internet. Many hardware solutions and software functions have been widely invented to avoid attacks. Packet classification and deep packet inspection (DPI) are operated in these systems to detect and protect potential threats by dropping harmful traffic. Packet classification acts as the initial filter to the network in which network traffic is classified into flows based on a pre-defined set of rules. It needs the inspection of multiple fields of the packet header and is different with IP forwarding where only the destination IP address is inspected.

When using hardware to implement packet classification solutions, memory used to store the pre-defined rules is often the bottleneck. Especially, for the platforms like Field Programmable Gate Arrays (FPGAs), on-chip memory is limited. Using external memory will cause lower clock rate because of the connection delay between two devices. To conquer these problems, numerous solutions have been proposed in the literature to reduce the memory usage of ruleset storage. Most of the existing papers take advantages of some rule properties to reach the goal of memory efficiency [3]–[7]. While these properties may not always exist, different ruleset might affect the results. In some cases, the heavily feature-dependent solutions may yield poor memory efficiency.

The solutions that consider memory efficiency and high throughput demand at the same time are difficult to implement and challenging due to many reasons. For example, in trie or tree based approaches, on-chip resources are easily exhausted due to pipelined tree traversal and multi-field lookup. Therefore, it will be difficult to implement multiple parallel pipelines to improve the throughput. In this work, we consider improving throughput as the primary goal.

Software Defined Networking (SDN) has been proposed as an innovative architecture for enterprise networks. SDN separates the software based control plane from the hardware based data plane, and uses a flexible protocol - OpenFlow [1] to manage network traffic. One of the most important functions in OpenFlow platforms is the flow table lookup [8], [9]. The flow table lookup needs to check the incoming packet to see if it is matched against multiple fields in a prioritized

flow table. The method is similar to the classic 5-field packet classification [10].

Many existing solutions for multi-field packet classification are implemented by using ternary content addressable memories (TCAMs) [11], [12]. But TCAMs are expensive and power hungry. TCAM based solutions also suffer from range explosion problem when converting ranges into prefixes [12]. In [23], an extended TCAM was proposed to solve the range explosion problem by introducing an iterative structure of 1-bit range check circuit.

Field Programmable Gate Array (FPGA) technology has been used to implement IP lookup, packet classification and other solutions for real time network processing [13], [14]. FPGA based packet classification engine can achieve very high throughput for rule sets of moderate size [15]. We can increase the throughput by creating numerous parallel pipelines. However, as the number of packet header fields or the rule set size increases, FPGA based approaches still suffer from clock rate degradation. Because OpenFlow protocol has many packet header fields to be examined [14], OpenFlow packet classification remains a challenging research problem.

II. RELATED WORK

A. FIELD-SPLIT BIT VECTOR (FSBV)

The Field-Split Bit Vector (FSBV) algorithm is designed to solve packet classification problems [3]. One of the most important things for FSBV is to reduce memory consumption. The studies on the 5-field traditional packet classification rules appearing in the Snort [16] ruleset show that the source address, destination address, and protocol fields of the rules contain a pretty small number of unique values compared with the ruleset size. Because reducing the requirement of memory is the major goal in FSBV, the authors applied the field-splitting algorithm only to the source port and destination port fields.

In FSBV, a given field F of width w bits can be split to a set of w sub-fields $F[w_i]$ for $i = 0$ to $w - 1$. Each w_i has two possible values, 0 and 1. Extending this to all the N rules in the ruleset will result in two N-bit vectors (say $R_0[N]$ and $R_1[N]$) that correspond to the two possible input values of sub-field $F[w_i]$. If bit j is set in $R_0[N]$, sub-field $F[w_i]$ of rule j is zero or don't care. Similarly, if bit j is set in $R_1[N]$, sub-field $F[w_i]$ of rule j is 1 or don't care. Therefore, if the input sub-field $F[w_i]$ is zero, we will examine $R_0[N]$; otherwise we will examine $R_1[N]$. We use an example ruleset to explain this process. We set the header field bit length w to 4. The bit vector generation method and packet lookup process are illustrated in Figure 1.

When packet header arrives, the corresponding field F will be used to fetch the related bit vectors. These bit vectors will be sent to a bit-wise logical AND to compute matching results. A bit vector in this packet classification algorithm consists of N bits. Each bit represents a rule of the ruleset. A bit in a bit vector is set to 1 to indicate a match or 0 to indicate a mismatch. The correctness of this method has already been proved in [3]. The result of the bit-wise AND operation is also an N-bit bit vector, where bit position k indicates if the rule k is a match for the incoming packet header or not.

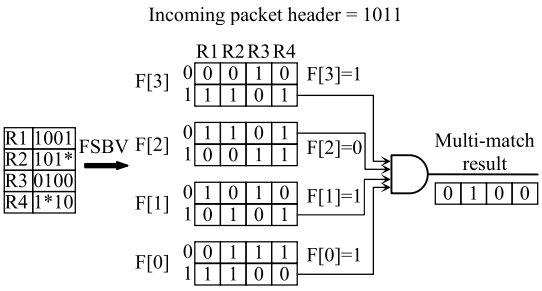

FIGURE 1. **Field-Split Bit Vector (FSBV).**

This field split algorithm is fully supported for wildcard (*) matching. The method to support wildcards is to set the corresponding bit vector to 1. In this example, wildcards appear in rule R2 and R4, and the positions are at 0 and 2, respectively. A wildcard means that the corresponding bit in the header can be either 0 or 1. Therefore, both the bits in positon 0 for R2 and in position 2 for R4 are set to 1.

Since the field split method does not support ranges, direct range-to-prefix conversion is usually used. However, the direct range-to-prefix conversion suffers a problem that a single rule will be partitioned into multiple rules.

B. STRIDE BIT VECTOR (StrideBV)

As mentioned earlier, the field split algorithm is only applied for source port and destination port fields due to the consideration of memory consumption. For the other fields, TCAM/CAM is used since the number of unique values is small. Their goal is to find out a solution that avoids relying on ruleset features and achieves high throughput. Implementing TCAM on FPGA can be inefficient due to high circuit complexity and poor performance compared with pipelined architectures [6]. Implementing TCAM on FPGA limits the scalability as well as performance of FSBV as ruleset features change. The author considers memory consumption as a secondary target mainly because FPGA has various memory resources. Hence, the proposed field split algorithm in [15] was applied to all the 5 fields.

In the case of traditional 5-field packet classification, using the original FSBV method will result in 104 stages in a single pipeline. This will cause serious latency problem. From a hardware perspective, numerous stages will cause significant routing delay, which leads to clock rate degradation. Reducing the pipeline stages can be done by using multiple bits instead of a single bit. This can be performed by storing bit vectors corresponding to the 2^k combinations of the k bit stride and loading a single bit vector per stage. The authors call it StrideBV scheme. StrideBV uses more memory while it has lower memory bandwidth to achieve high throughput. To be specific, for a rule containing the prefixes of W bits, using stride d requires $2^d \times \lceil W/d \rceil$ bits.

C. OpenFlow

The Open Networking Foundation (ONF), a user-led organization dedicated to promotion and adoption of software-defined networking (SDN), manages the OpenFlow standard. ONF defines OpenFlow as the first standard communications interface defined between the controls and forwarding layers of an SDN architecture. OpenFlow allows direct access to and manipulation of the forwarding plane of network devices such as switches and routers, both physical and virtual (hypervisor-based). It is the absence of an open interface to the forwarding plane that has led to the characterization of today's networking devices as monolithic, closed, and mainframe-like. A protocol like OpenFlow is needed to move network control out of proprietary network switches and into control software that's open source and locally managed.

An OpenFlow Switch consisting of one or more flow tables and a group table performs packet lookups and forwarding via a secure OpenFlow channel to an external controller. The switch communicates with the controller and the controller manages the switch via the OpenFlow protocol. Based on the OpenFlow protocol, the controller can add, update, and delete flow entries in flow tables, both reactively (in response to packets) and proactively. Each flow table in the switch contains a set of flow entries and each flow entry consists of match fields, counters, and a set of instructions to apply to the matching packets.

Matching process starts at the first flow table and may continue to additional flow tables. Flow entries match packets in the order of increasing priority, with the first matching entry in each table being used. If a matching entry is found, the instructions associated with the specific flow entry are executed. If no match is found in a flow table, the outcome depends on configurations: to forward the packet to the controller over the OpenFlow channel, to drop the packet, or to continue to the next flow table.

OpenFlow version 1.0 packet header includes 12 fields: Ingress port, Ethernet source address, Ethernet destination address, Ethernet type, VLan ID, VLan priority, IP source address, IP destination address, IP protocol, IP ToS bits, Transport source port/ICMP Type, Transport destination port/ICMP Type. Each entry contains a specific value, prefix, or don't care. Details on the properties of each field are described in Table 1.

OpenFlow-compliant switches come in two types: OpenFlow-only, and OpenFlow-hybrid. OpenFlow-only switches support only OpenFlow operations. In those switches, all packets are processed by the OpenFlow pipeline, and cannot be processed otherwise.

OpenFlow-hybrid switches support both OpenFlow operation and normal Ethernet switching operation, i.e. traditional L2 Ethernet switching, VLAN isolation, L3 routing (IPv4 routing, IPv6 routing...), ACL and QoS processing. Those switches must provide a classification mechanism outside of OpenFlow that routes traffic to either the OpenFlow pipeline or the normal pipeline. For example, a switch may use the VLAN tag or input port of the packet to decide whether to process the packet using one pipeline or the other, or it may direct all packets to the OpenFlow pipeline. An OpenFlow-hybrid switch may also allow a packet to go from the OpenFlow pipeline to the normal pipeline through the NORMAL and FLOOD reserved ports.

TABLE 1. Field lengths of Openflow entries.

Field	Bits
Ingress Port	6
Ethernet source address	48
Ethernet destination address	48
Ethernet type	16
VLAN id	12
VLAN priority	3
IP source address	32
IP destination address	32
IP protocol	8
IP ToS bits	6
Transport source port/ICMP Type	16
Transport destination port/ICMP Code	16
Total # of bits	243

D. FIELD PROGRAMMABLE GATE ARRAY

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence ''field-programmable''. The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an application-specific integrated circuit (ASIC).

Contemporary FPGAs have large resources of logic gates and RAM blocks to implement complex digital computations. FPGAs can be used to implement any logical function that ASICs could perform. The ability to update the functionality after shipping, partial re-configuration of the design, and the low non-recurring engineering costs relative to an ASIC design offers advantages for many applications.

Historically, FPGAs have been slower, less energy efficient and generally achieved less functionality than their fixed ASIC counterparts. More recently, FPGAs such as the Xilinx Virtex-7 or the Altera Stratix 5 have come to rival corresponding ASIC and ASSP solutions by providing significantly reduced power, increased speed, lower materials cost, minimal implementation real-estate, and increased possibilities for re-configuration 'on-the-fly'.

Vendors can take a middle road by developing their hardware on ordinary FPGAs, but manufacture their final version as an ASIC so that it can no longer be modified after the design has been committed.

FPGA can be used to solve any problem which is computable. This is trivially proven by the fact FPGA can be used to implement a soft microprocessor. Their advantage lies in that they are sometimes significantly faster for some applications due to their parallel nature and optimality in terms of the number of gates used for a certain process.

III. PROPOSED SCHEME

Our proposed scheme is based on the bit vector scheme [15]. A fixed number of rules are implemented in a pipelined architecture. Therefore, if the rule set is large, multiple pipelines are needed. In each pipeline, the incoming packet's header first passes through several processing stages for range fields. The partial bit vector results of range field stages will be passed to the following stages of the prefix fields. Finally, a pipelined priority encoder will aggregate multiple pipeline results and find out the highest priority match. We use the OpenFlow version 1.0, and we set the ingress port to 6 bits, so the incoming packet header is of size 243 bits. Our design takes the advantage of the statistic results of [18] with consideration of reasonable number of stages.

A. RANGE BIT VECTOR ENCODING (RBVE) SCHEME

As the traditional packet classification rule tables, OpenFlow rule tables only have the fields of source and destination ports that contain 16-bit ranges. A 16-bit range is denoted by [*LB*, *UB*], where *LB* and *UB* are its lower and upper bounds. In this section, we propose a new range encoding scheme called *range bit vector encoding* (*RBVE*) scheme. In RBVE scheme, a 16-bit range is split into many d-bit sub-ranges, where d is the fixed stride size that can be 1, 2, 4, or 8. Note that the cases of variable stride sizes are not discussed in this paper. For a fixed stride size of d, there are s d-bit sub-ranges that can be implemented as a pipeline of s stages, where $s = 16/d$. Figure 2 illustrates the general view of a 16-bit range [*LB*, *UB*] being split into four 4-bit sub-ranges, $[LB_i, UB_i]$ for $i = 1$ to 4. Let A be the 16-bit input address and A is split into four sub-addresses, A_i for $i = 1$ to 4. As we can see, if $UB_1 < A_1 < LB_1$, we can conclude that $UB < A < LB$ and so the first stride is sufficient to decide if the input address A matches the range [*LB*, *UB*]. Figure 2 also shows the other three possible conditions, (1) $A_1 > UB_1 | A_1 < LB_1$, (2) $LB_1 < UB_1 = A_1$, and (3) $A_1 = LB_1 < UB_1$. Another additional condition not shown in Figure 2 is $A_1 = LB_1 = UB_1$.

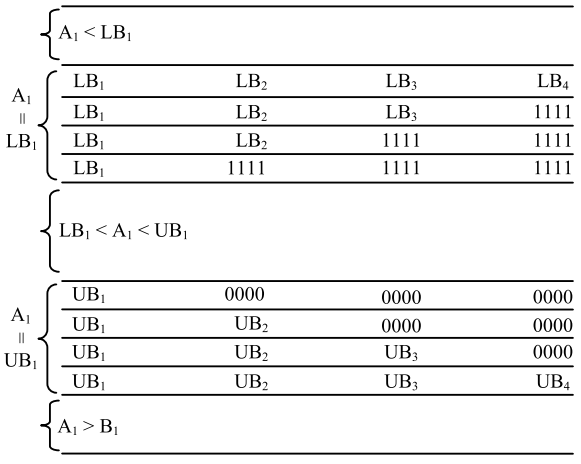

FIGURE 2. General view of a range being split into 4 strides.

111: match, not depending on the following stage;
001: match, only depending on LB of the following stage;
100: match, only depending on UB of the following stage;
010: match, depending on LB and UB of the following stage;
000: mismatch, not depending on the following stage;

FIGURE 3. Meaning of the output signals from each stage.

Therefore, we propose to use a set of output signals that are generated from each stage to perform the matching process. Figure 3 shows these output signals and their meanings. Take the first stride as an example. Output signals 000 and 111 mean always mismatch and always match for the two conditions $A_1 > UB_1 | A_1 < LB_1$ and $LB_1 < A_1 < UB_1$, respectively. Output signal 001 means $A_1 = LB_1 < UB_1$ and so a match happens only when $A_2 \circ A_3 \circ \ldots \circ A_s \geq LB_2 \circ LB_3 \circ \ldots \circ LB_s$, where $\circ$ is the concatenation operator. Similarly, output signal 100 means $LB_1 < UB_1 = A_1$ and so a match happens when $A_2 \circ \ldots \circ A_s \leq UB_2 \circ \ldots \circ UB_s$. Output signal 010 means $A_1 = LB_1 = UB_1$ and so a match happens only when $LB_2 \circ \ldots \circ LB_s \leq A_2 \circ \ldots \circ A_s \leq UB_2 \circ \ldots \circ UB_s$.

Initialize $a_2a_1a_0 = 000$;
If $(LB_1 < A_1 < UB_1)$ $a_2a_1a_0 = 111$; // *special case*
else
  if $(UB_1 > LB_1$ and $A_1 = UB_1)$ $a_2 = 1$;
  if $(A_1 = UB_1 = LB_1)$ $a_1 = 1$;
  if $(UB_1 > LB_1$ and $A_1 = LB_1)$ $a_0 = 1$;

(a)

Initialize $c_3c_2c_1c_0 = 0000$;
if $(A_i = UB_i)$ $c_3 = 1$; if $(A_i = LB_i)$ $c_1 = 1$;
if $(A_i < UB_i)$ $c_2 = 1$; if $(A_i > LB_i)$ $c_0 = 1$;

(b)

Initialize $f_1f_0 = 00$;
if $(A_s \leq UB_s)$ $f_1 = 1$.
if $(LB_s \leq A_s)$ $f_0 = 1$.

(c)

FIGURE 4. RBVE code design with the fixed stride of d bits for the range fields, where $s = 16/d$ and A_i $(i = 1..s)$ is any d-bit address. (a) The 3-bit code ($Code_1[A_1] = a_2a_1a_0$) for the first stage. (b) The 4-bit code ($Code_i[A_i] = c_3c_2c_1c_0$) for the middle stages. (c) The 2-bit code ($Code_s[A_s] = f_1f_0$) for the last stage.

Figure 4 shows the detailed code design for the RBVE scheme. In RBVE scheme, 3-bit, 4-bit, and 2-bit codes are used in the first, middle, and the last stages, respectively. There are 2^d possible input addresses denoted by A_i for the d-bit sub-ranges. Given the stride with lower and upper bounds LB_i and UB_i for $i = 1$ to s where $s = 16/d$, we pre-compute the 3-bit codes of the first stage, the 4-bit codes of the middle stages, and the 2-bit codes of the last stages, for all the possible input addresses.

Case	$Code_1(A_1)$	Description
	$a_2a_1a_0$	Condition 1: $LB_1<UB_1$
$A_1>UB_1$	000	Always mismatch, not depending on $Code_2(A_2)$.
$A_1=UB_1$	100	Depend on $Code_2(A_2)$
$LB_1<A_1\&A_1<UB_1$	111	Always match, not depending on $Code_2(A_2)$.
$A_1=LB_1$	001	Depend on $Code_2(A_2)$
$A_1<LB_1$	000	Always mismatch, not depending on $Code_2(A_2)$.
Condition 2: $UB_1=LB_1$		
$A_1>UB_1$	000	Always mismatch, not depending on $Code_2(A_2)$.
$A_1=UB_1$	010	Depend on $Code_2(A_2)$
$A_1<LB_1$	000	Always mismatch, not depending on $Code_2(A_2)$.

(a)

Case	$Code_2(A_2)$	Description
	f_1f_0	Condition 1: $LB_2 < UB_2$
$A_2 > UB_2$	01	Match if $Code_1(A_1)$=001.
$LB_2\le A_2\&A_2\le UB_2$	11	Match if $Code_1(A_1)$=010/001/100.
$A_2 < LB_2$	10	Match if $Code_1(A_1)$=100.
Condition 2: $UB_2 < LB_2$		
$A_2 \ge LB_2$	01	Match if $Code_1(A_1)$=001.
$UB_2<A_2\&A_2<LB_2$	00	Always mismatch
$A_2 \le UB_2$	10	Match if $Code_1(A_1)$=100.
Condition 3: $UB_2 = LB_2$		
$A_2 > UB_2$	01	Match if $Code_1(A_1)$=001.
$A_2 = UB_2$	11	Match if $Code_1(A_1)$=010/001/100.
$A_2 < LB_2$	10	Match if $Code_1(A_1)$=100.

(b)

FIGURE 5. Code details for range split into two sub-ranges. (a) $Code_1(A_1)$. (b) $Code_2(A_2)$.

1) CODE DESIGN FOR $d = 8$

In order to understand the proposed code design easily, we first assume $d = 8$. Thus, only the first and the last stages are needed because the range fields are 16 bits. Figure 4(a) and 4(c) show the codes for these two stages. Let LB_1 (LB_2) and UB_1 (UB_2) be the lower and upper bounds of the first (last) stride of the range [LB, UB]. Similarly, let A_1 (A_2) be the first (last) stride value of the input address. In the first stage, LB_1 must be smaller than or equal to UB_1. As shown in Figure 4(a), code $a_2a_1a_0 = 111$ is used for the special case of $LB_1 < A_1 < UB_1$, where the first stage is sufficient to know the input address matches range [LB, UB]. Other than this special case, bits a_2, a_1, and a_0 are set to one for the cases of $A_1 = UB_1 < LB_1$, $A_1 = UB_1 = LB_1$, and $UB_1 < LB_1 = A_1$, respectively. The rationale of enabling a_2, a_1, and a_0 is as follows. These three bits are designed to be mutually exclusive. Therefore, at most one of these three bits can be set to one, except the above special case. For the case of $A_1 = UB_1 > LB_1$, we set $a_2 = 1$. A match is only possible when the following stride of the input address satisfies the condition of $A_2 \le UB_2$. Therefore, we set $f_1 = 1$ when the address of the last stride $A_2 \le UB_2$ as shown in Figure 4(c). If both a_2 and f_1 are set to one simultaneously, we generate a final match result. Similarly, a_0 is set to 1 when $UB_1 > LB_1 = A_1$ and $f_0 = 1$ when $LB_2 \le A_2$. Thus, if both a_0 and f_0 are one, a final match is generated. For the case of $A_1 = UB_1 = LB_1$, a match occurs only when $LB_2 \le A_2 \le UB_2$. Therefore, a_1, f_1, and f_0 are all set to one. In addition to the cases considered above, the other two cases happening at the first stage are $A_1 > UB_1$ and $A_1 < LB_1$. For these two cases, the initial code 000 is used. It is not possible to have a match result for these two cases no matter what the code for the last stride is.

The code designs of the first and last stages can be understood clearly based on the relationship between UB_1 and LB_1 and between LB_2 and UB_2. Given a range [LB, UB], the relationship between UB_1 and LB_1 only satisfies two conditions: $UB_1 > LB_1$ and $UB_1 = LB_1$ as shown in Figure 5(a). In other words, the condition of $UB_1 < LB_1$ will never occur. However, the relationship between LB_2 and UB_2 meets all three possible conditions, $UB_2 > LB_2$, $UB_2 = LB_2$, and $UB_2 < LB_2$ as shown in Figure 5(b). By using these conditions, all the possible cases are listed in Figure 5(a) and (b) and their associated codes can be given easily.

After reading the codes $Code_1[A_1] = a_2a_1a_0$ in the first stage and $Code_s[A_2] = f_1f_0$ in the last stage, we still have to perform a simple computation to determine if it is a match or mismatch by the equation shown in Figure 6(c).

$u_2u_1u_0 = a_2a_1a_0$;

(a)

$u_2u_1u_0 =$
111 if $((b_2\&b_1\&b_0) \mid (b_2\&c_2) \mid (b_1\&c_2\&c_0) \mid (b_0\&c_0))$; (e1)
100 if $((b_2\&c_3) \mid (b_1\&c_3\&!c_1))$; (e2)
001 if $((b_0\&c_1) \mid (b_1\&!c_3\&c_1))$; (e3)
010 if $(b_1\&c_3\&c_1)$; (e4)

(b)

$match = ((b_2\&b_1\&b_0) \mid ((b_2\&f_1) \mid (b_1\&f_1\&f_0) \mid (b_0\&f_0))$; (e5)

(c)

FIGURE 6. Output signals of the *RBVE* scheme with stride $d = 8$, 4, and 2, where $b_2b_1b_0$ is the input signals. (a) Output signals ($u_2u_1u_0$) of the first stage. (b) Output signals ($u_2u_1u_0$) of the middle stages. (c) Output signal (*match*) of the last stage.

2) CODE DESIGN FOR $d = 4$ or 2

The codes for the first and the last stages are the same as the case of $d = 8$. The codes for the middle stages are shown in Figure 4(b). Since all the three conditions of $A_i < UB_i$, $A_i = UB_i$, $A_i > UB_i$ are possible in the middle stages, we need two bits (c_3 and c_2) to distinguish them. We use c_3 and c_2 independently. Bit c_3 is enabled when $A_i = UB_i$ and c_2 is enabled when $A_i < UB_i$. Thus, c_3 and c_2 cannot be enabled simultaneously. Similarly, we need two bits (c_1 and c_0) to record the relationship between A_i and LB_i. Bit c_1 is enabled when $A_i = LB_i$ and c_0 is enabled when $A_i > LB_i$. By following the same design rationale of bits $a_2a_1a_0$ at the first stage, we have to perform some computations to obtain the 3-bit output signals ($u_2u_1u_0$) as shown in Figure 6. As a result, the

LB: 10000110									
UB: 10101011									
Stride	1	2	3	4	5	6	7	8	
0	00	11	01	11	01	10	00	01	
1	11	00	10	00	10	01	11	10	

FIGURE 7. Code example of range [134, 171] with 1-bit stride.

2-bit code *ab* for every stage.
a = !*UB*;
b = !*LB*;

FIGURE 8. Code design for range fields with 1-bit stride.

process of determining the final match in the last stage will be same as the one shown in Figure 6(c).

3) CODE DESIGN FOR $d = 1$

The code design for $d = 1$ can be the same as that for $d = 8, 4$ or 2. However, this way seems wasting a lot of memory space since 1-bit stride is much simpler than the multibit stride. Therefore, we propose a memory efficient code design for $d = 1$ as follows. As defined before, $[LB_i, UB_i]$ is the sub-range in stride i. We know that LB_i or UB_i can only be 0 and 1. Therefore, the sub-range in the first stride can be [0, 0], [0, 1], or [1, 1]. The sub-range in the other strides can be [0, 0], [0, 1], [1, 0], or [1, 1]. The sub-address of stride i, A_i, can be 0 or 1. Similar to StrideBV scheme, we can store 2-bit codes xy for input address A_i based on the following pre-computation rule: (1) if $A_i = UB_i$, x is to 1; otherwise x is set to 0, and (2) if $A_i = LB_i$, y is to 1; otherwise y is set to 0. Figure 7 shows an example. Since the stored code xy for $A_i = 0$ is the complement of that for $A_i = 1$, we only store the former. As a result, we propose to store the code ab as shown in Figure 8.

With the stored codes ab, the matching process is the same as the design for $d = 8, 4$ or 2 by using the output signals defined in Figure 3. Figure 9 shows the matching process at each stage.

First stage
010 if $((A_1{=}1\&ab{=}00) \mid (A_1{=}0\&ab{=}11))$;
001 if $(A_1{=}0\&ab{=}01)$;
100 if $(A_1{=}1\&ab{=}01)$;

Middle stages
111 if $((p_2p_1p_0{=}111) \mid (p_0\&A_i\&b) \mid (p_2\&!A_i\&!a))$;
010 if $(p_1\&A_i\&ab{=}00) \mid (p_1\&!A_i\&ab{=}11))$;
001 if $((p_1\&!A_i\&ab{=}01) \mid (p_0\&A_i\&!b) \mid (p_0\&!A_i\&b))$;
100 if $((p_1\&A_i\&ab{=}01) \mid (p_2\&A_i\&!a) \mid (p_2\&!A_i\&a))$;

Last stage
$match = ((p_2p_1p_0{=}111) \mid ((p_0\&b) \mid (p_0\&A_s\&!b)) \mid$
$((p_1\&A_s\&!a) \mid (p_1\&!A_s\&b)) \mid ((p_2\&!a) \mid (p_2\&!A_s\&a)))$;

FIGURE 9. Output signals ($u_2u_1u_0$) for 1-bit stride, where $p_2p_1p_0$ is input signals from previous stage, and A_i is the input address.

B. SEQUENTIAL SUBRANGE COMPARE (SSC) SCHEME

We propose a range matching scheme called sequential subrange compare (SSC) scheme that stores subranges directly in the memory and performs subrange match operations sequentially. SSC does not perform precomputations against input headers as in the bit vector based schemes. SSC is similar to iterative structure of 1-bit range check circuit of the extended TCAM [23]. The advantage of SSC over the extended TCAM is that SSC is more general than 1-bit range check circuit in that the stride size of SSC can be any number of bits. The original intention is to solve the range field processing errors occurring in the other schemes [18], [22] since all the subranges split from an original 16-bit range are not totally independent. The matching process of a subrange cannot be done by using only two comparators, e.g., the $\leq$ lower bound comparator and the $\geq$ upper bound comparator. A correct and complete implementation is much more complex. The subranges split from a range are classified into three types, the first, the middle, and the last subranges. Inferred from Figure 2, we need four types of matching results that are represented by the signals, *Match*, *Mismatch*, *LBmatch*, and *UBmatch*. *Mismatch* can be deduced from *Match*, *LBmatch*, and *UBmatch*. When none of *Match*, *LBmatch*, and *UBmatch* is true, *Mismatch* becomes true. Therefore, *Mismatch* is not needed. Also, *Match* and *LBmatch*/*UBmatch* are mutually exclusive and so they cannot be true simultaneously. However, *LBmatch*, and *UBmatch* may be true at the same time because lower and upper bounds can be the same. Let the range be split into s subranges, $[LB_i, UB_i]$ and the input address A be split into s sub-addresses A_i for $i = 1$ to s. Therefore, each stage only has to compute three signals using the signals generated from previous stage, the subrange bounds LB_i and UB_i, and the input sub-address A_i. The final stage only computes the signal *Match* that will be the final match result.

Figure 10 shows the pseudo codes that compute the three required signals. The match process at the first stage is

First stage 1
If $(A_1 > LB_1$ & $A_1 < UB_1)$ $\boldsymbol{Match_1 = 1}$ (f1)
If $(A_1 = LB_1)$ $\boldsymbol{LBmatch_1 = 1}$ (f2)
If $(A_1 = UB_1)$ $\boldsymbol{UBmatch_1 = 1}$ (f3)

Middle stage i ($i = 2$ *to* $s - 1$)
If ($Match_{i-1}$ or (g1)
$LBmatch_{i-1}$ & $UBmatch_{i-1}$ & $A_i > LB_i$ & $A_i < UB_i$ or (g2)
$LBmatch_{i-1}$ & !$UBmatch_{i-1}$ & $A_i > LB_i$ or (g3)
!$LBmatch_{i-1}$ & $UBmatch_{i-1}$ & $A_i < UB_i$) $\boldsymbol{Match_i = 1}$ (g4)
If $(LBmatch_{i-1}$ & $A_i = LB_i)$ $\boldsymbol{LBmatch_i = 1}$ (g5)
If $(UBmatch_{i-1}$ & $A_i = UB_i)$ $\boldsymbol{UBmatch_i = 1}$ (g6)

Last stage s
If ($Match_{s-1}$ or (h1)
$LBmatch_{s-1}$ & $UBmatch_{s-1}$ & $A_s \geq LB_s$ & $A_s \leq UB_s$ or (h2)
$LBmatch_{s-1}$ & !$UBmatch_{s-1}$ & $A_s \geq LB_s$ or (h3)
!$LBmatch_{s-1}$ & $UBmatch_{s-1}$ & $A_s \leq UB_s$) $\boldsymbol{Match_s = 1}$ (h4)

FIGURE 10. Pseudo code of the proposed *SSC* scheme.

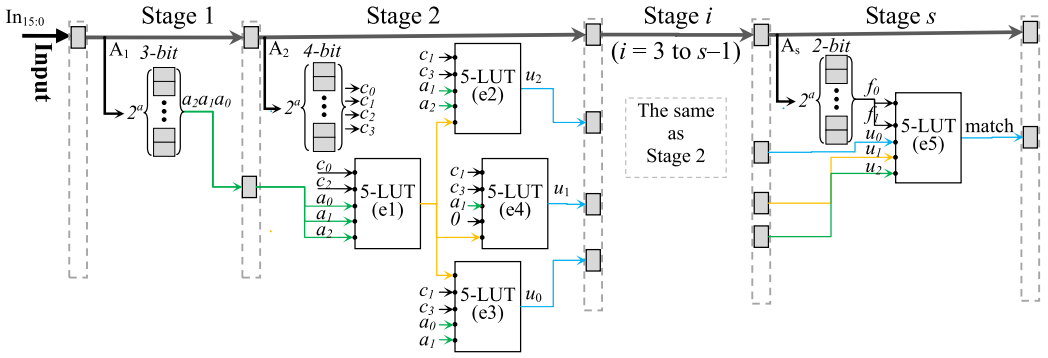

FIGURE 11. Range field pipeline of the *RBVE* scheme with stride size *d*.

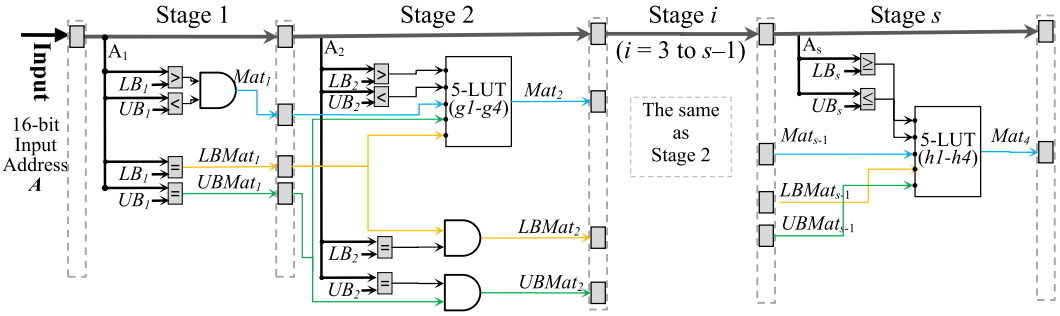

FIGURE 12. Hardware range implementation of the *SSC* scheme.

simple, as shown in equations (f1)-(f3). For the middle stages, the *Match* signal is true when one of the four cases represented by equations (g1)-(g4) happens. Equation (g1) similar to (f1) considers when the sub-addresses preceding the stage is the same as both the lower and upper bounds of the corresponding subrange and the sub-address is between the lower and upper bounds of the subrange at the current stage. Equation (g3) considers the case in which the sub-addresses preceding the stage is the same as the lower, not upper bounds of the corresponding subrange and the sub-address is larger than the lower of the subrange at the current stage. Equation (g4) is similar to equation (g3). Consider the 4-stage example in Figure 2 and assume the current middle stage is 3. Equation (g2) states that when $A_1 = LB_1 = UB_1$ and $A_2 = LB_2 = UB_2$, the condition of $LB_3 < A_3 < UB_3$ indicates a final match. Equation (g3) states that when $A_1 \circ A_2 = LB_1 \circ LB_2$ but $A_1 \circ A_2 \neq UB_1 \circ UB_2$, the condition of $LB_3 < A_3$ indicates a final match due to $LB < A < UB$, where $\circ$ is the concatenation operator. For equations (g5) and (g6), the conditions of $A_1 \circ \ldots \circ A_i = LB_1 \circ \ldots \circ LB_i$ and $A_1 \circ \ldots \circ A_i = UB_1 \circ \ldots \circ UB_i$ enable the signals $LBmatch_i$ and $UBmatch_i$ at stage i, respectively. The last stage is in fact the same as the middle stages, except that the enabled $LBmatch_i$ and $UBmatch_i$ signals are also considered as a match. As a result, equations (h1)-(h4) modify equations (g1)-(g4) by replacing the $>$ and $<$ comparators with $\geq$ and $\leq$ comparators.

C. HARDWARE IMPLEMENTATION OF RBVE AND SSC

The pipeline implementations of RBVE and SSC schemes are given in Figure 11 and Figure 12, respectively. These two figures show the required resources for a single 16-bit range that is split into $s = 16/d$ stages of d-bit sub-ranges, where $d = 8$, 4, 2, or 1. For the *RBVE* scheme in Figure 11, 5-input LUTs (denoted by 5-LUTs) are used to implement the logic of all the stages. A memory array of 2^d 3-bit, 4-bit, and 2-bit entries is needed for the first, the middle, and the last stages, respectively. Notice that the middle stages are not needed when $d = 8$. The values stored in these memory arrays are the precomputed codes for all possible 2^d d-bit sub-addresses. For the *SSC* scheme in Figure 12, only two d-bit memory spaces are needed for a sub-range to store its lower and upper bounds. However, four comparators are required in each stage. Each comparator for d-bit strides is implemented by a $2d$-input LUT.

1) BLOCK RAM STORAGE

The block RAM in Xilinx®6 and 7 series FPGAs stores up to 36 Kbits of data and can be configured as either two independent 18 Kb RAMs, or one 36 Kb RAM. Since the stride size d of the proposed *RBVE* scheme is not larger than 8 for the range fields, the basic unit of the block memory we use is the 18Kb block RAM which is configured as a 512×36 memory unit (i.e., an array of 512 36-bit entries)

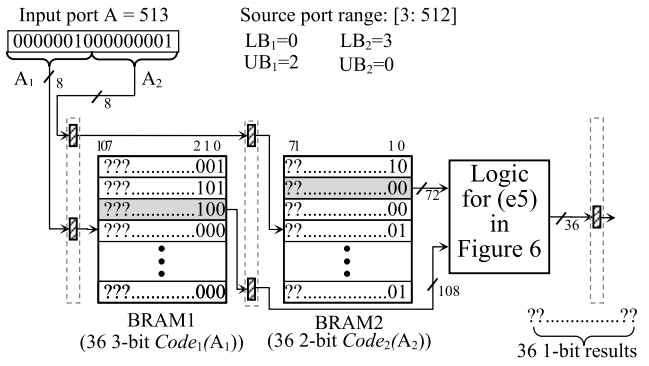

FIGURE 13. **The block RAM layout for *RBVE* scheme.**

in true dual-port mode. In the true dual-port mode, two concurrent reads on the same block RAM can be performed and so the throughput of the search operations can be doubled.

Since the proposed *RBVE* scheme uses the bit array to store the pre-computed initial matching results, the block RAM is a good implementation choice. Also, the minimum number of entries for a block RAM is 512, the proper choice of the stride size d is 8, which requires only 256 entries. As a result, a half of the block RAM becomes wasted. We will address this problem later. Figure 13 shows the proposed block RAM layout for *RBVE* scheme with stride size $d = 8$. The pre-computed codes of 36 ranges are concatenated for satisfying the minimum block RAM configuration of 512 36-bit entries. BRAM1 and BRAM2 are the block RAMs of 256 entries needed for $\text{Code}_1(\text{A}_1)$ and $\text{Code}_2(\text{A}_2)$ at the stages 1 and 2, respectively. As shown in the figure, the rightmost code of these 36 codes is for the range [3: 512]. So, for the input port number 513, the match signal for this range is 0, which is a mismatch.

2) DISTRIBUTED RAM STORAGE

The function generators (LUTs) in SLICEMs of the Xilinx®6 and 7 series FPGAs can be implemented as a synchronous RAM resource called a *distributed RAM* denoted by *distRAM*. The other type of slices in Xilinx®6 and 7 series FPGAs is SLICEL that is usually 2-3 times larger than SLICEM. SLICEL cannot be configured as distRAM. Multiple 6-input LUTs in a SLICEM can be combined in various ways to store larger amount of data as follows, single-port distRAM of $32 \times 1/64 \times 1/128 \times 1/256 \times 1$ bits, dual-port distRAM of $32 \times 1/64 \times 1/128 \times 1$ bits, and quad-port distRAM of $32 \times 2/64 \times 1$ bits. Specifically, four 6-input LUTs can be configured as one single-port distRAM of 256×1 bits, or one dual-port distRAM of 128×1 bits, or one quad-port distRAM of 64×1 bits.

For choosing smaller stride size while still considering reasonable number of stages, we choose stride size $d = 4$ for the range fields of the proposed SSC scheme. Another advantage of choosing $d = 4$ is the comparators of decreasing order of rule priorities. Therefore, this priority

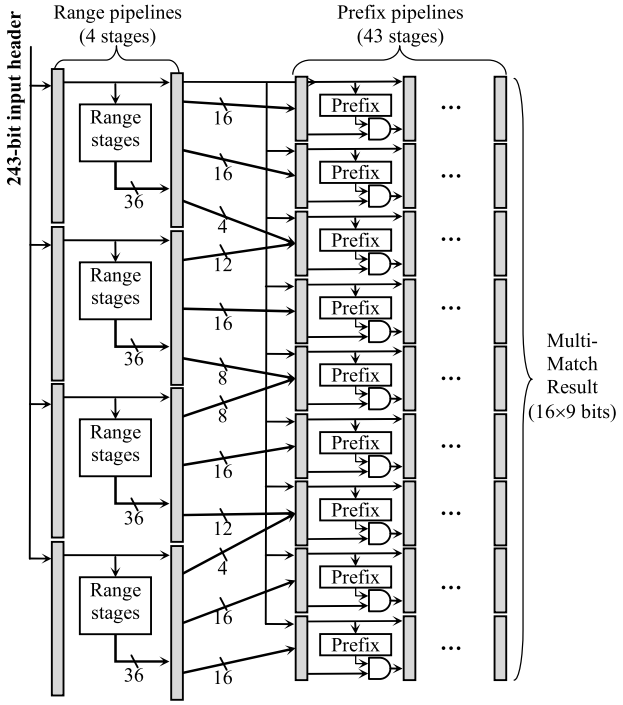

FIGURE 14. **A super-pipeline of range pipelines using *RBVE* and prefix pipelines using *StrideBV* with cross-pipeline links.**

two 4-bit inputs can be implemented with one 4-input and one 5-input LUTs. The cost of implementing comparators with two d-bit inputs for $d > 4$ will be too expensive. So, the number of stages for a 16-bit range field is 4. As a result, each range field value in the SSC scheme takes eight 4-bit memory space to store the lower and upper bounds of four 4-bit sub-ranges. Sixteen range values are grouped in a cluster and so that the distributed RAM of 8×16 bits is needed in each of the four stages for a 16-bit range field.

D. PREFIX FIELD MATCHING

We use *StrideBV* method for the prefix field stages. Assume the stride size is k. So, we will have $\lceil (243 - 32)/k \rceil$ stages for all prefix fields. For $k \leq 8$, we can use distRAM and 16 rules are grouped in a cluster. For $k > 8$, we can use block RAM and 36 rules are grouped in a cluster. The prefix field stages are placed after the range field stages and so the partial N-bit bit vector result of range field will be passed in, where $N = 36$ for *RBVE* scheme and $N = 16$ for *SSC* scheme.

Figure 14 shows an overall architecture of the range and prefix field implementations by *RBVE* and *StrideBV* schemes, respectively. The *RBVE* scheme for range fields is implemented by using block RAMs, where stride size is 8 bits and 36 ranges are grouped together in a range pipeline as shown in Figure 14. The *StrideBV* scheme for prefix fields is implemented by using distRAMs, where stride size is 5 bits and 16 prefixes are grouped together in a prefix pipeline. Since the sizes of the groups in the range and prefix fields

are different, cross-pipeline links are needed as shown in the figure. We call it the *super-pipeline* that combines the range and prefix pipelines. As a result, totally 144 rules are in a super-pipeline when all fields are considered. We refer to this super-pipeline architecture as *Design I*. If the range fields are implemented by distRAM based on SSC scheme, we use 16 ranges in a group which is the same as the prefix fields implemented by StrideBV scheme. Therefore, no cross-pipeline link is needed. We refer to this architecture as *Design II*. Table 2 summarizes the implementations of Design I and Design II.

TABLE 2. Comparisions of design I and II.

Implementation	Ranges	Prefixes
Design I	RBVE scheme Stride = 8 bits blockRAM 36 ranges/group	StrideBV Stride = 5 bits distRAM 16 ranges/group
Design II	SSC scheme Stride = 4 bits distRAM 16 ranges/group	

In order to process more rules, we need to implement multiple pipelines. Restricted by the on-chip resources, the total number of rules that can be supported by design I is limited by the size of the block RAM. However, total number of rules that can be supported by design II is limited by the size of the distributed RAM.

E. IMPROVING THE UTILIZATION OF BLOCK RAMs

As stated earlier, a half amount of the block RAM is wasted for range fields that are implemented by the *RBVE* scheme. We propose a simple indexing scheme to avoid wasting any block RAM. This indexing scheme is based on the observation that all the prefix fields except the source and destination IPs in the Openflow tables contain only the singleton values. In other words, $(211 - 64) = 147$ bits of a rule must not be don't care. Therefore, it is possible to select one bit (say bit x) out of these 147 bits such that the number of rules with bit $x = 0$ is roughly equal to that with bit $x = 1$. If it is not possible, we can select more bits (say n bits) to divide all the rules into 2^n segments. Then it is simple to find m out of 2^n segments such that the number of rules in these m segments is roughly equal to that in the other $2^n - m$ segments. Based on this indexing scheme, we can always divide the rules into two groups of the same size. The real implementation can be done by an additional stage. For a block RAM of 512 entries, the rules in the first group use the upper 256 entries and the rules in the other group use the lower 256 entries. Since one or more out of the 211 bits in the prefix fields are consumed in the additional stage of the indexing scheme, the number of stages in the prefix fields will be reduced at least one stage. As a result, the overall number of the stages after applying the proposed indexing scheme will not grow. Specifically, if the indexing scheme picks one bit, the number of prefix stages reduces from 43 to 42.

F. PIPELINED PRIORITY ENCODER

The pipelined priority encoder is to aggregate multiple pipelines together to find out the final matched rule. The input of pipelined priority encoder is multiple multi-match results produced by each pipeline. And the output is the ID of the single highest-priority matched rule. Because the number of pipelines is often large, this encoder uses the pipeline to reduce clock period. The rules are arranged from top to bottom in the decreasing order of rule priorities. Therefore, this priority encoder outputs the index of the first set bit from top in the bitmap of the matching results. The design of the priority pipeline for the set of 6K rules is shown in Figure 15. Figure 16 shows the overview of the complete pipeline architecture.

IV. EXPERIMENTAL RESULTS

We conducted experiments using Xilinx ISE Design Suite 14.7 where speed grade is set to −2. The device used

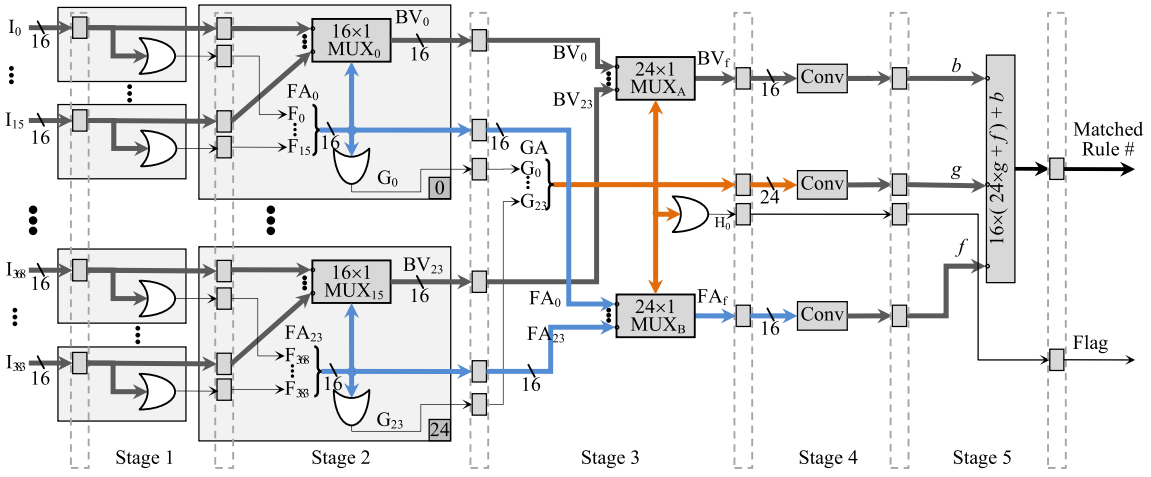

FIGURE 15. The pipelined priority encoder of 6K rules.

TABLE 3. Performance of the proposed schemes.

XC6VLX760	Slices Registers (948,480)	SLICEM (132,480 LUTs = 8Mb single-port distRAM)			SLICEL (341,760 LUTs)			Block RAMs (720×36Kb)	Bonded IOBs (1,200)	Through -put (MHz)
		Prefix	Range	total	PriEnc	logic	Total			
Design I 6K Single port	433,276(45%)	126K (2^5×42×6K bits)	N/A	126K (99%) (2^5×42×6K bits)	3.2K	68.5K	71.7K	256×5×2×6 Kb (59%)	257(21%)	283
Design I 3K Dual port	433,276 (45%)	126K (2^5×42×3K bits)	N/A	126K (99%) (2^5×42×3K bits)	3.2K	70.5K	73.7K	256×5×2×3K (30%)	266(22%)	566
Design II 5K	195,291(20%)	107.5K (2^5×43×5K bits)	10K (7.7%) (64×2×5Kbits)	117.5K (94%) (7520K bits)	3.2K	139.5K	142.7K	N/A	252(21%)	380

TABLE 4. Performance comparisons.

Approach	Throughput (MHz)	Latency (Clocks)	# of Fields Supported	Speed Depending on ruleset	Range Field Support	# of Rules Supported
Design I-dual port	566	53	12	No	Yes	3K
Design II	380	57	12	No	Yes	5K
Scalable Packet Classification [21]	125	36	12	Yes	No	1K
High-performance architecture [18]	373	72	15	No	Yes*	1K
StrideBV [15]	235	30	5	No	No	0.5K
Scalable and Modular Architecture [22]	526	28	5	No	Yes*	28K

*: Range field is supported but incomplete

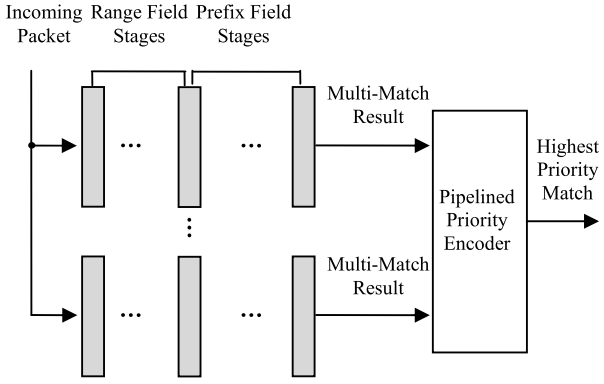

FIGURE 16. Multi-pipeline System Overview.

in the experiments is Xilinx®6, Virtex-6 XC6VLX760 [19]. Virtex-6 XC6VLX760 has 1200 I/O pins, 25,920 Kb BRAM (720 36Kb BRAM blocks), 118,560 logic slices where SLICEM can be configured to realize a large distributed RAM (up to 8,280 Kb). Each slice contains four 6-input LUTs and 8 flip-flops. The numbers of rules in Design I and Design II are 6K/3K (single/dual port) and 5K, respectively.

The resource utilization and performance statistics of the Design I and II are shown in Table 3. The SLICEM that can be configured as distRAM limits the size of rule tables in the implementation. In Design I with single port, one bit from 211 prefix field bits is used for the block RAM utilization improvement stated in section III.D. As a result, only 42 stages for prefix fields are needed. Utilizing 99% of the distRAM provided in FPGA device can support up to 6K rules. In addition, the block RAM for range fields takes 59%. In Design I with double port, the number of supportable rules is 3K because double ported distRAM doubles the usage of LUTs in SLICEM. However, the throughput can be doubled. In Design II, only the distRAM can be used because the constraint of 512 entries in block RAM is hard to overcome. We set the stride of range fields to four because large distributed RAM causes serious clock degradation. Compared to Design I, the additional distRAM used for range fields in Design II is 128 bits per rule. Totally, 5K rules can be supported.

We will compare our proposed designs with other related work based on StrideBV algorithm or OpenFlow rule sets as shown in Table 4. Depending on ruleset characteristic, the performance of the scheme proposed in [21] varies because it uses decision tree based method. Others use StrideBV algorithm and so the performance will not vary. Scalable and Modular Architecture proposed in [22] supports 5-field rules and can achieve very high throughput. To our knowledge, our proposed designs are the first range supported method that can achieve the throughput of 380 MHz. Method in [18] can also sustain a high throughput but its range field design is not complete and thus cannot always get correct results. Methods in [22] also support a range processing solution. The proposed designs are also much faster than the original paper [15] and even store 10 times larger ruleset.

V. CONCLUSION

In this paper, we proposed two designs for processing the range fields. The proposed *RBVE* scheme for range fields is similar to the *StrideBV* scheme in that both use a

special encoding to store the pre-computed results in memory. The proposed SSC scheme is a simple sub-range comparison method that using pipeline to sequentially find matching result. By performing experiments on Xilinx Virtex-6 XC6VLX760 FPGA device, the proposed designs can achieve the throughputs of 380 MHz and 566 MHz based on single-ported and dual-ported memory.

REFERENCES

[1] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.
[2] OpenFlow Foundation. *OpenFlow Switch Specification, Version 1.0.0*. [Online]. Available: http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf, accessed Jun. 24, 2015.
[3] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using FPGAs," in *Proc. Symp. Parallelism Algorithms Archit. (SPAA)*, 2009, pp. 188–196.
[4] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proc. ACM SIGCOMM*, 1999, pp. 147–160.
[5] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM*, 2003, pp. 213–224.
[6] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *Proc. ACM/SIGDA FPGA*, 2005, pp. 238–245.
[7] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, Jan./Feb. 2000.
[8] OpenFlow Foundation. *OpenFlow Switch Specification, Version 1.1.0*. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources / onf - specifications/openflow/openflow-spec-v1.1.0.pdf, accessed Jun. 24, 2015.
[9] OpenFlow Foundation. *OpenFlow Switch Specification Version 1.3.1*. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf, accessed Jun. 24, 2015.
[10] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Netw.*, vol. 15, no. 2, pp. 24–32, Mar./Apr. 2001.
[11] F. Yu, R. H. Katz, and T. V. Lakshman, "Efficient multimatch packet classification and lookup with TCAM," *IEEE Micro*, vol. 25, no. 1, pp. 50–59, Jan./Feb. 2005.
[12] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," in *Proc. ACM SIGCOMM*, 2005, pp. 193–204.
[13] S. Yi, B.-K. Kim, J. Oh, J. Jang, G. Kesidis, and C. R. Das, "Memory-efficient content filtering hardware for high-speed intrusion detection systems," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2007, pp. 264–269.
[14] A. Majumdar, S. Cadambi, M. Becchi, S. T. Chakradhar, and H. P. Graf, "A massively parallel, energy efficient programmable accelerator for learning and classification," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 6:1–6:30, Mar. 2012.
[15] T. Ganegedara and V. K. Prasanna, "StrideBV: Single chip 400G+ packet classification," in *Proc. IEEE 13th Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2012, pp. 1–6.
[16] Snort. *Snort: Network Intrusion Prevention and Detection System (IPS/IDS)*. [Online]. Available: http://www.snort.org/, accessed Jun. 24, 2015.
[17] T. Sasao, "On the complexity of classification functions," in *Proc. 38th Int. Symp. Multiple Valued Logic (ISMVL)*, May 2008, pp. 57–63.
[18] Y. R. Qu, S. Zhou, and V. K. Prasanna, "High-performance architecture for dynamically updatable packet classification on FPGA," in *Proc. ANCS*, Oct. 2013, pp. 125–136.
[19] Xilinx. *7 Series FPGAs Overview*. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf, accessed Jun. 24, 2015.
[20] Xilinx. *Virtex-6 Family Overview*. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, accessed Jun. 24, 2015.
[21] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 9, pp. 1668–1680, Sep. 2012.
[22] T. Ganegedara, W. Jiang, and V. K. Prasanna, "A scalable and modular architecture for high-performance packet classification," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 5, pp. 1135–1144, May 2013.
[23] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs," in *Proc. IEEE Int. Conf. Netw. Protocols (ICNP)*, Nov. 2003, pp. 120–131.

YEIM-KUAN CHANG received the Ph.D. degree in computer science from Texas A&M University, College Station, in 1995. He is currently a Professor with the Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan. His research interests include Internet router design, computer architecture, and multiprocessor systems.

CHUN-SHENG HSUEH received the M.S. degree in computer science and information engineering from National Cheng Kung University, Tainan, Taiwan, in 2014. His research interests include high-speed packet processing in hardware and deep packet inspection architectures.