High-Performance and Dynamically Updatable Packet Classification Engine on FPGA

Yun R. Qu, *Member, IEEE* and Viktor K. Prasanna, *Fellow, IEEE*

Abstract—High-performance and dynamically updatable hardware architectures for multi-field packet classification have regained much interest in the research community. For example, software defined networking requires 15 fields of the packets to be checked against a predefined rule set. Many algorithmic solutions for packet classification have been studied over the past decade. FPGA-based packet classification engines can achieve very high throughput; however, supporting dynamic updates is yet challenging. In this paper, we present a two-dimensional pipelined architecture for packet classification on FPGA; this architecture achieves high throughput while supporting dynamic updates. In this architecture, modular Processing Elements (PEs) are arranged in a two-dimensional array. Each PE accesses its designated memory locally, and supports prefix match and exact match efficiently. The entire array is both horizontally and vertically pipelined. We exploit striding, clustering, dual-port memory, and power gating techniques to further improve the performance of our architecture. The total memory is proportional to the rule set size. Our architecture sustains high clock rate even if we scale up (1) the length of each packet header, or/and (2) the number of rules in the rule set. The performance of the entire architecture does not depend on rule set features such as the number of unique values in each field. The PEs are also self-reconfigurable; they support dynamic updates of the rule set during run-time with very little throughput degradation. Experimental results show that, for a 1 K 15-tuple rule set, a state-of-the-art FPGA can sustain a throughput of 650 Million Packets Per Second (MPPS) with 1 million updates/second. Compared to TCAM, our architecture demonstrates at least four-fold energy efficiency while achieving two-fold throughput.

Index Terms—Packet classification, field-programmable gate array (FPGA), two-dimensional pipeline, dynamic updates

1 Introduction

Software Defined Networking (SDN) has been proposed as a novel architecture for enterprise networks. SDN separates the software-based control plane from the hardware-based data plane; as a flexible protocol, OpenFlow [1], [3] can be used to manage network traffic between the control plane and the data plane. One of the kernel function Open-Flow performs is the flow table lookup [1]. The flow table lookup requires multiple fields of the incoming packet to be examined against entries in a prioritized flow table. This is similar to the classic multi-field packet classification mechanism [4]; hence we use interchangeably the flow table lookup and the OpenFlow packet classification in this paper.

The major challenges of packet classification include: (1) supporting large rule sets, (2) sustaining high performance [2], and (3) facilitating dynamic updates [5]. Many existing solutions for multi-field packet classification employ Ternary Content Addressable Memories (TCAMs) [6], [7]. TCAMs cannot support efficient dynamic updates; for example, a rule to be inserted can move across the entire rule set [8]. This is an expensive operation. TCAMs are not scalable with respect to the rule set size. Besides, they are also very power-hungry [2], [9], [10].

• *The authors are with the Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089. E-mail: {yunqu, prasanna}@usc.edu.*

Manuscript received 17 Aug. 2014; revised 18 Nov. 2014; accepted 2 Jan. 2014. Date of publication 7 Jan. 2015; date of current version 16 Dec. 2015. Recommended for acceptance by A. Gordon-Ross.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2389239

Field Programmable Gate Array (FPGA) technology has been widely used to implement algorithmic solutions for real-time applications [11], [12]. FPGA-based packet classification engine can achieve very high throughput for rule sets of moderate size [13]. However, as the number of packet header fields or the rule set size increases (e.g., OpenFlow packet classification [1]), FPGA-based approaches often suffer from clock rate degradation.

Future Internet applications require the hardware to perform frequent incremental updates and adaptive processing [14], [15], [16]. Because it is prohibitively expensive to reconstruct an optimal architecture repeatedly for timely updates, many sophisticated solutions have been proposed for packet classification supporting dynamic updates over the years [5]. Due to the rapid growth of the network size and the bandwidth requirement of the Internet [17], it remains challenging to design a flexible and run-time reconfigurable hardware-based engine without compromising any performance.

In this paper we present a scalable architecture for packet classification on FPGA. The architecture consists of multiple self-reconfigurable Processing Elements (PEs); it sustains high performance for packet classification on a large number of packet header fields. This architecture also supports efficient dynamic updates of the rule set. The rule set features, the size of the rule set, and the packet header length all have little effect on the performance of the architecture. Our contributions in this work include:

- *Scalable architecture*. A two-dimensional pipelined architecture on FPGA, which sustains high throughput even if the length and the depth of the packet classification rule set are scaled up.

TABLE 1
An Example of OpenFlow Packet Classification Rule Set (5 Rules, 15 Fields) [1], [2]

RID	Priority	Ingr	Metadata	Eth_src	Eth_dst	Eth_type	VLAN_ID	VLAN_priority	MPLS_label	MPLS_tfc	SA	DA	Prtl	ToS	SP	DP
Field Length		32	64	48	48	16	12	3	20	3	32	32	8	6	16	16
Field Type†		E	E	E	E	E	E	E	E	E	P	P	E	E	E	E
R_0	2	5	*	00:13:A9:00:42:40	00:13:08:C6:54:06	0x0800	*	5	0	*	001*	*	TCP	0	*	*
R_1	1	*	*	08:00:69:02:FC:07	00:FF:FF:FF:FF:FF	*	100	7	16,000	0	00*	1011*	UDP	*	*	*
R_2	3	*	*	*	00:00:00:00:00:00	0x8100	4,095	7	*	*	1*	1011*	*	*	2	5
R_3	4	1	*	00:FF:FF:FF:FF:FF	*	*	4,095	*	*	*	1*	1*	*	0	7	5
R_4	0	4	*	FF:FF:FF:FF:FF:FF	*	*	2,041	*	*	*	110*	01*	*	0	80	123

†: *"E" as exact match, and "P" as prefix match.*

- *Novel optimization techniques*. A couple of optimization techniques exploiting tradeoffs between various design parameters and performance metrics, including rule set size, throughput, update rate, resource consumption, and power efficiency.
- *Distributed update algorithms*. A set of algorithms supporting dynamic updates, including modify, delete and insert operations on the rule set. The algorithms are performed distributively on self-reconfigurable PEs. The update operations have little impact on the sustained throughput.
- *Superior throughput*. Detailed performance evaluation of our proposed architecture on a state-of-the-art FPGA. We show in post-place-and-route results that our architecture sustains a throughput of 650 MPPS with 1 million updates/second (M updates/s) for a 1 K 15-tuple rule set.
- *Energy efficiency*. Thorough comparison of our architecture with existing solutions for packet classification with respect to energy efficiency. Compared to TCAM, our architecture sustains $2\times$ throughput and supports fast dynamic updates with $4\times$ energy efficiency.

The rest of the paper is organized as follows: Section 2 introduces the classic multi-field packet classification problem and its OpenFlow variant. We revisit existing packet classification techniques in Section 3. We detail the design of the two-dimensional pipelined architecture in Section 4. Optimization techniques are proposed in Section 5. We present the update algorithms on this architecture in Section 6. Section 7 provides the experimental results and summarizes the advantages of the proposed architecture. Section 8 concludes the paper.

2 Background

2.1 Classic Packet Classification

An individual predefined entry used for classifying a packet is denoted as a *rule*; a rule is associated with a unique rule ID (RID) and a priority. The data set consisting of all the rules is called *rule set*. Packet classification can be defined as: *Given a packet header having d fields and a rule set of size N, out of all the rules matching the packet header, report the RID of the rule whose priority is the highest.*

A packet header is considered to be matching a rule only if it matches all the fields of that rule. The classic packet classification [4] involves five fields in the packet header: the Source/Destination IP Addresses (SA/DA), Source/Destination Port numbers (SP/DP), and the transport layer protocol (Prtl). Note different fields in a rule can require different types of match criteria; thus, a rule can consist of prefixes, exact values, or/and ranges. In this paper, we consider prefix match in the SA and DA fields, and exact match in all the other fields.

2.2 OpenFlow Packet Classification

OpenFlow packet classification requires a larger number of packet header fields to be examined. For example, in the current specification of OpenFlow protocol [1], a total number of 15 fields consisting of 356 bits in the packet header have to be compared against all the rules in the rule set. Out of the 15 fields, only the SA and DA fields require prefix match, while all the other fields require exact match. We show an example rule set in Table 1. A "*" in a particular field of a rule indicates the corresponding rule can match any value in that field.

As shown in Table 2, compared to the classic packet classification, it is more challenging to achieve high performance for OpenFlow packet classification:

1) *Large-scale*. OpenFlow requires a large number of bits and packet header fields to be processed for each packet (large d and large L).
2) *Dynamic updates*. OpenFlow places high emphasis on dynamic updates (large U), including rule modification, deletion, and insertion.

3 Prior Work

3.1 Packet Classification Techniques

Packet classification has been extensively studied over the past decade [4]. Most of the packet classification algorithms

TABLE 2
Classic versus OpenFlow Packet Classification

Type	Classic	OpenFlow
No. of fields (d)	5	15
Pkt. header length (L)	104 bits [13]	356 bits [1]
Update rate (U)	Relatively static	$\leq$10 K updates/s [8]

used in hardware or software fall into two major categories: decision-tree-based [10], [18] and decomposition-based [5], [19] algorithms.

Decision-tree-based approaches involve cutting the search space recursively into smaller subspaces based on the information from one or more fields in the rule. In [10], a decision tree is mapped onto a pipelined architecture on FPGA; for a rule set containing 10 K rules, a throughput of 80 Gbps is achieved for packets of minimum size (40 bytes). However, 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 [18]). As a result, a decision-tree can use up to $O(N^d)$ memory; this approach can be impractical.

Decomposition-based approaches [19], [20] first search each packet header field individually. The partial results are then merged to produce the final result. To merge the partial results from all the fields, hash-based merging techniques [20] can be explored; however, these approaches either require expensive external memory accesses, or rely on a second hardware module to solve hash collisions.

As a decomposition-based approach, Bit Vector (BV) approach [21] is a specific technique in which the lookup on each field returns an N-bit vector. Each bit in the bit vector corresponds to a rule. A bit is set to one only if the input matches the corresponding rule in this field. A bit-wise logical AND operation can be exploited to gather the matches from all the fields.

For decomposition-based approaches, the complexity of searching a specific field is usually dependent on some rule set features, such as the number of unique rules in a field [19], [20]. In this paper, we propose an approach whose performance does not depend on such rule set features.

3.2 BV-Based Approaches

The Field-Split BV (FSBV) approach [21] and its variants [13] split each field into multiple *subfields* of s-bits; a rule is mapped onto each subfield as a ternary string defined on $\{0, 1, *\}^s$. Lookup operations can be performed in all the subfields in a pipelined fashion; the partial result in each PE[1] is represented by a BV of N bits. Logical AND operations can be used to merge all the extracted BVs to generate the final match result on FPGA. We show the basic architecture [13], [21] of BV-based approaches in Fig. 1; FSBV approach can be visualized as a special case of $s = 1$. To access an N-bit data, wires of length $O(N)$ are often used for the memory; as N increases, the clock rate of the entire pipeline deteriorates.

3.3 Dynamic Updates

Dynamic updates for packet classification has been a well-defined problem [5]; we are not aware of any solution supporting high performance. In [5], two algorithms are proposed based on tree/trie structures to support dynamic updates; they require $O(\log^d N)$ and $O(\log^{d+1} N)$ update time, respectively, for a d-field rule set consisting of N rules. They are too expensive for OpenFlow packet classification ($d = 15$). For the same reason, most of the decision-tree-based approaches cannot easily support fast dynamic updates. Some of the decision-tree-based approaches [10] require the trees to be recomputed and remapped onto FPGA whenever the rule set needs to be updated; this is very expensive.

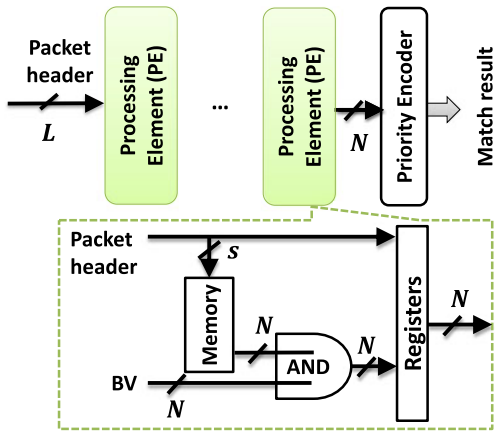

Fig. 1. Basic architecture of BV-based approaches.

Some of the decomposition-based approaches [20] explore external memory on FPGA; for each update, a number of external memory write accesses must be performed. This is also very expensive. In this paper, we construct self-reconfigurable PEs, and we propose an efficient update scheme using these PEs.

4 Scalable Architecture

4.1 Notations

For a packet header of L bits, we split all the fields into subfields of s bits[2]; hence there are in total $\lceil \frac{L}{s} \rceil$ subfields, indexed by $j = 0, 1, \ldots, \lceil \frac{L}{s} \rceil - 1$. Let k_j denote the input packet header bits in subfield j; therefore the length of k_j is also s bits. A *bit vector* $B_j^{(k_j)}$ is defined as the vector specifying the matching conditions between the input packet header bits and the corresponding bit locations of all the rules. We show an example in Fig. 2, where BVs are the column vectors indexed by k_j in subfield j.

For a rule set consisting of N rules, the length of each BV is N bits. We denote the data structure that stores all the BVs in a given subfield as *bit vector array*, as shown in Fig. 2. In this figure, each 2-bit stride is associated with an $N \times 2^s = 3 \times 4$ BV array. An $N \times 2^s$ BV array requires a memory size of $2^s \times N$.

After we have constructed all the bit vectors in all the subfields, we use the input header bits k_j's to address the corresponding BVs in the BV arrays. For a subfield j, $B_j^{(k_j)}$ is extracted for the input bits k_j. For example, in Fig. 2, if the input packet header has $k_0 = 10$ in the subfield $j = 0$, we extract the BV $B_{0,10} = 010$; this indicates only the rule R_1 matches the input in this subfield.

4.2 Challenges and Motivations

As shown in Fig. 1, in the pipelined architecture for BV-based approaches on FPGA, each PE extracts a BV in a

1. We use PE to denote a pipeline stage that produces a BV.

2. This is different from FSBV, where only two fields are split.

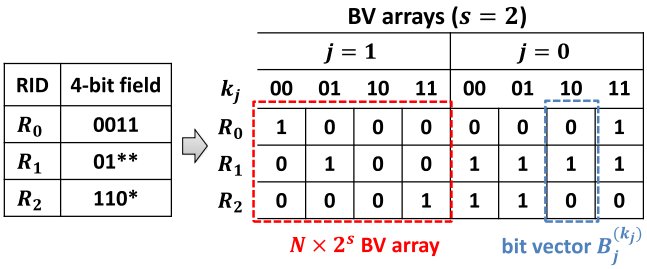

Fig. 2. Splitting a 4-bit field into two subfields.

subfield; to produce the final match result, BVs are ANDed with each other in a pipelined fashion. Excluding the priority encoder (if any), we have $\lceil \frac{L}{s} \rceil$ PEs in this pipeline. We denote such an architecture as *basic pipelined architecture*. In the basic pipelined architecture, the BVs in each PE for N rules are N-bit long. For distributed RAM (distRAM) or Block RAM (BRAM) modules of fixed size[3], the number of memory modules required for each PE grows linearly with respect to 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.

To address this problem, we propose a two-dimensional pipelined architecture consisting of multiple modular PEs in this work. Our work is inspired by the observation that a long BV can be further partitioned into smaller *subvectors* to eliminate the long wires and improve the overall clock rate of the classification engine.

4.3 Modular PE

A modular PE is used to match a single packet header against one rule ($N = 1$) in a 1-bit subfield ($s = 1$). In order to minimize the number of I/O pins utilized, a modular PE is also responsible for propagating input packet headers to other PEs. A modular PE should be able to handle both prefix match and exact match.

Let us consider the internal organization of a *modular PE* as shown in Fig. 3. The main difference between the modular PE in Fig. 3, and the PEs used in the basic pipelined architecture, is that the modular PE in Fig. 3 only produces the match result for exactly one rule at a time (2×1-bit data memory). The modular PE in this work has other components:

1) Rule decoder. The logic-based rule decoder is mainly responsible for dynamic updates (See Section 6).
2) s-bit register for input packet header. It is used in the construction of vertical pipelines (See Section 4.4). Note there are two types of registers: We denote the register buffering the input packet header bits as the *input register*; we denote the register after the AND gate as the *output register*.

We denote a rule requiring prefix match as a *prefix rule*. Similar to FSBV, for a 1-bit subfield, the prefix rule can be handled efficiently. In Fig. 3, the packet header bit is used to address the memory directly; the extracted BV is then ANDed with the BV output from the previous PE in the pipeline.

3. e.g., $2^6 \times 1$-bit distRAM based on six-input Look Up Table (LUT) and 36 Kb SRAM-based BRAM [22].

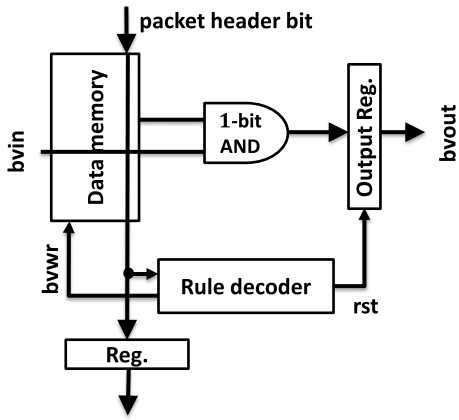

Fig. 3. Modular PE.

A rule requiring exact match can be treated as a special case of a prefix rule. Hence we do not introduce any other new components for exact match.

4.4 Two-Dimensional Pipeline

To handle a larger number of rules and more input packet header bits, we use multiple modular PEs to construct a complete two-dimensional pipelined architecture as shown in Fig. 4. We use PE$[l, j]$ to denote the modular PE located in the lth row and jth column, where $l = 0, 1, 2, 3$ and $j = 0, 1, 2$. We use distRAM for the data memory in each PE, so that the overall architecture can be easily fit on FPGA and the memory access in each PE is localized.

4.4.1 Horizontal

We define horizontal direction as the forward (right) or backward (left) direction along which the BVs are propagated. We use output registers of modular PEs to construct horizontal pipelines (e.g., PE$[0, 0]$, PE$[0, 1]$, PE$[0, 2]$). The data propagated in the horizontal pipelines mainly consist of BVs.

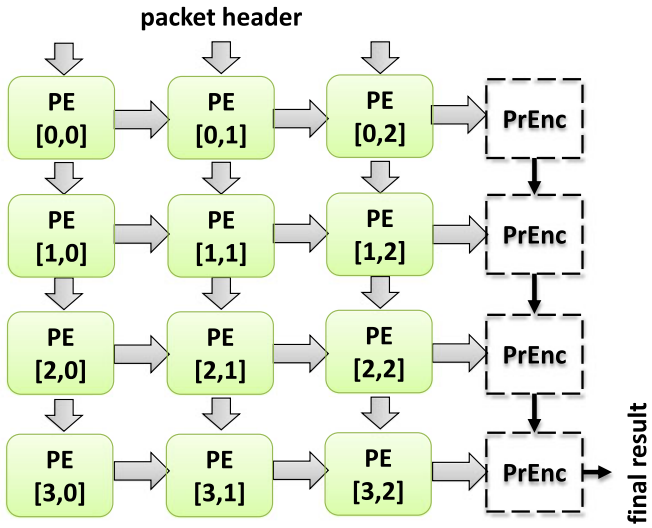

Fig. 4. Example: A two-dimensional pipelined architecture ($N = 4$, $L = 3$) and priority encoders (PrEnc).

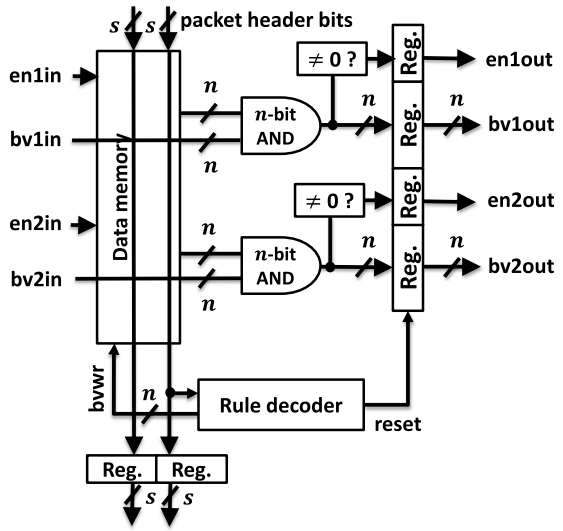

Fig. 5. A modular PE with striding, clustering, and power gating techniques; data memory is dual-ported.

4.4.2 Vertical

We define vertical direction as the upward (up) or downward (down) direction along which the input packet header bits are propagated. We use input registers of modular PEs to construct vertical pipelines (e.g., PE$[0,0]$, PE$[1,0]$, PE$[2,0]$, PE$[3,0]$). The data propagated in the vertical pipelines of PEs consist of packet header bits.

Note we do not restrict the rules to be arranged in the rule set following any specific order, we need a priority encoder [13] at the end of each horizontal pipeline to report the highest-priority match. In our approach, the match results of all the horizontal pipelines are collected by a vertical pipeline of priority encoders.

Since each modular PE can perform prefix/exact match for one rule in a 1-bit subfield, the architecture in Fig. 4 consisting of four rows and three columns of modular PEs can handle four rules, each rule having three 1-bit subfields. Using more modular PEs, this architecture can be scaled up for a larger number of rules, and for longer packet headers. For a rule set consisting of N rules, and an L-bit packet header, the two-dimensional pipelined architecture requires N rows and L columns of PEs to be concatenated in a pipelined fashion.

5 Optimization Techniques

5.1 Striding

The striding technique [13] can be applied to the modular PE, as shown in Fig. 5. Suppose the modular PE only needs to perform packet header match against one rule. Based on the discussion in Section 4.1, the amount of memory required for prefix match in an s-bit subfield is $2^s \times 1$. The length of the input register is s bits because we have s input packet header bits when using striding technique.

5.2 Clustering

Besides the striding technique, we also introduce a *clustering* technique for the modular PE. The basic idea is to build a PE which can handle multiple rules instead of a single rule. Let us consider the modular PE performing packet header match against n rules as shown in Fig. 5. We construct a BV array consisting of 2^s bit vectors, each of length n; this requires a data memory of size $2^s \times n$. The 1-bit AND gate in Fig. 3 is adjusted to an n-bit logical AND gate in Fig. 5.

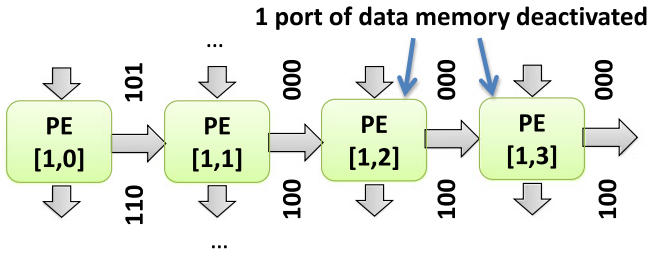

Fig. 6. Power gating.

5.3 Dual-Port Data Memory

We employ dual-port (read) data memory on FPGA. Two concurrent packets can be processed in each modular PE. We denote the input BVs for the two concurrently processed packets as $bv1in$ and $bv2in$, respectively; we denote the output BVs for the two concurrent packets as $bv1out$ and $bv2out$, respectively. The throughput is twice the maximum clock rate achieved on FPGA. Assuming the same clock rate can be sustained, this technique doubles the throughput achieved by the modular PE shown in Fig. 3.

We show a modular PE with dual-port data memory techniques in Fig. 5. For each of the two concurrent packets, the modular PE compares an s-bit subfield of the packet header against a set of n rules. The overall two-dimensional pipelined architecture has $\lceil \frac{N}{n} \rceil$ rows by $\lceil \frac{L}{s} \rceil$ columns of PEs; each row of PEs (i.e., a horizontal pipeline) is responsible for matching packet headers against n rules, while each column (i.e., a vertical pipeline) is in charge of matching an s-bit subfield.

5.4 Power Gating

For any incoming packet header, if PE$[l,j]$ ($j < \frac{L}{s} - 1$) identifies that this packet header does not match any of the n rules, then there is no need to examine the remaining subfields for this packet. This is because the logical AND operations in PE$[l,j+1]$, PE$[l,j+2], \ldots,$ and PE$[l,\frac{L}{s}-1]$ can only produce all-"0" vectors anyway; the outputs from the data memory modules become irrelevant for these PEs. By deactivating the corresponding port of the data memory in PE$[l,j+1]$, PE$[l,j+2], \ldots,$ and PE$[l,\frac{L}{s}-1]$, a significant amount of power can be saved on the average (see Section 7.7). We show the modular PE with power gating technique in Fig. 5. The comparator after each AND gate generates an enable signal; this enable signal is high only if the corresponding output BV is not an all-"0" vector. The enable signal (high/low) is output to the next PE in order to activate/deactivate the corresponding port of the data memory in the next PE.

For example in Fig. 6, suppose $bv1out$ and $bv2out$ are "000" and "100" after PE$[1,1]$, respectively. For any modular PE after PE$[1,1]$ (namely PE$[1,2]$ or PE$[1,3]$), the logical AND gates can only produce all-"0" vectors as their $bv1out$ signals. In this case, the data memory ports corresponding to $bv1out$ in both PE$[1,2]$ and PE$[1,3]$ are deactivated. Note

in this example, the data memory ports corresponding to $bv2out$ are never deactivated.

6 Dynamic Updates

6.1 Problem Definition

OpenFlow packet classification requires the hardware to adapt to frequent incremental updates for the rule set during run-time. In this section, we propose a dynamic update scheme which supports fast incremental updates of the rule set without sacrificing the pipeline performance. Before we detail our update mechanism, we define the following terms:

- *Old rule*. The rule to be modified in the rule set
- *New rule*. The rule to appear in the rule set after an update
- *Outdated (data structure)*. (Data structure, e.g. BV, BV array, and valid bit) to be updated
- *Up-to-date (data structure)*. (Data structure) that is already updated

Given a rule set Φ consisting of N rules $\{R_i | i = 0, 1, \ldots, N-1\}$, we reiterate the problem definition of dynamic updates as three subproblems:

- *Modify*. Given a rule with RID R, and all of its field values and priority, search RID R in Φ, locate $i \in \{0, 1, \ldots, N-1\}$ where $R_i = R$; change rule R_i in Φ into rule R (e.g., change the SA field of R_1 in Table 1 from "00*" into "0*").
- *Delete*. Given a rule with RID R, search RID R in Φ, locate $i \in \{0, 1, \ldots, N-1\}$ where $R_i = R$; delete rule R_i from Φ (e.g., remove R_0 completely from Table 1).
- *Insert*. Given a rule with RID R, and all of its field values and priority, search RID R in Φ; if $\forall i \in \{0, 1, \ldots, N-1\}$, $R_i \neq R$, then insert rule R into Φ (e.g., add a brand new rule as R_5 into Table 1).

We assume all modular PEs are implemented with striding and clustering techniques; each of the PEs stores an $n \times 2^s$ BV array.

Notice that the first step of all update operations is always a RID check, which reports whether the RID of the new rule exists in the current rule set. RID check only requires exact match for a $\log N$-bit field; the rule decoders in the first column of PEs are in charge of this process, since the results of the RID check need to be reported before any update operation.

After RID check is completed, to target the above three subproblems, we present our main ideas as follows:

- (Section 6.2) We update all the corresponding BVs in the BV arrays, or we update priority encoders (for priority).
- (Section 6.3) We keep a "valid" bit for each rule; we reset the bit to invalidate a rule.
- (Section 6.4) We check the valid bits of all the rules first; if a rule in the rule set is invalid, we modify this invalid rule into the new rule, and validate this new rule.

Section 6.5 covers the architectural support for the update mechanism. The resulting overall architecture consists of multiple self-reconfigurable PEs, each PE configuring its memory contents in a distributed manner. Section 6.6 summarizes the update schedule.

6.2 Modification

After the RID check, suppose RID R already exists in the rule set; hence $\exists i \in \{0, 1, \ldots, N-1\}$ such that $R_i = R$. Rule modification can be performed as: Given a rule with RID R, along with all of its field values and priority, compute the up-to-date BVs, and replace the outdated BVs in the BV arrays with the up-to-date BVs. In any subfield, we assume a rule is represented by a ternary string $\{0, 1, *\}^s$. In reality, a rule is represented by two binary strings, the first string specifying the non-wildcard ternary digits while the second string specifying the wildcards.[4]

6.2.1 Modifying Prefix Rule

Let us consider how to update the BV arrays. The first step for rule modification is to construct the up-to-date BVs for this subfield. Specifically, we use Algorithm 1 to construct all 2^s up-to-date bit vectors (of length n) for this s-bit subfield. The correctness of Algorithm 1 can be easily proved [21]. Note Algorithm 1 is a distributed algorithm; if the modification of rule R_i requires multiple BV arrays to be updated, Algorithm 1 is performed in parallel by the PEs in the same horizontal pipeline where R_i can be located. In each PE, the logic-based rule decoder performs Algorithm 1 to update the memory content by itself (self-reconfiguration).

Algorithm 1. Up-to-Date BVs in Subfield j

Input n ternary strings each of s bits: $T_{i,j}$, where $T_{i,j} \in \{0, 1, *\}^s$, $i = 0, 1, \ldots, n-1$.
Output 2^s BVs each of n bits: $B_j^{(k_j)} = \overline{b_{j,0}^{(k_j)} b_{j,1}^{(k_j)} \ldots b_{j,n-1}^{(k_j)}}$, where $b_{j,i}^{(k_j)} \in \{0, 1\}$, $k_j = 0, 1, \ldots, 2^s - 1$, and $i = 0, 1, \ldots, n-1$.

```
for i = 0, 1, ..., n − 1 do
    for k_j = 0, 1, ..., 2^s − 1 do
        if k_j matches T_{i,j} then
            b_{j,i}^{(k_j)} ← 1
        else
            b_{j,i}^{(k_j)} ← 0
        end if
    end for
end for
```

As shown in Fig. 2, the BVs are arranged in an orthogonal direction to the rules in the data memory. To modify a single rule, 2^s memory write accesses are required in the worst case.[5] As can be seen later, even in the worst case, no more than 2^s bits are modified in our approach.

We show an example for rule modification in Fig. 7. In this example, we modify the subfield $j = 0$ of the rule R_2 in Fig. 2. In this subfield, based on Algorithm 1, R_2 is to be updated from "0*" to "1*". A naive solution is to update the

4. e.g., a tenary string "01*" is represented by "010" and "001".

5. This happens when the outdated BV is different from the up-to-date BV in every single bit location. Also, we assume each time there is only one rule to be updated; updating multiple rules at the same time is not supported by the rule decoder.

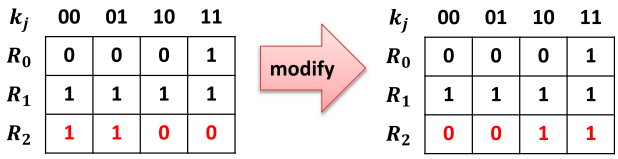

Fig. 7. Modifying R_2.

entire BV array. However, since we exploit distRAM for data memory, each bit of a BV is stored in one distRAM entry; this means every bit corresponding to a rule can be modified independently. Hence in Fig. 7, to update the subfield $j = 0$ of R_2, only 4 bits have to be modified (in four memory accesses). To avoid data conflicts, memory write accesses are configured as single-ported. Hence in any subfield, we always allocate 2^s clock cycles for 2^s memory write accesses (worst case) for simplicity.

6.2.2 Modifying Priority

If the update process requires the priority of the old rule to be changed, i.e., the new rule and the old rule have different priorities, we update the priority encoders based on a dynamic tree structure [23]. The time complexity to update the dynamic tree is $O(\log N)$. In general, if a prioritized rule set requires prefix match to be performed, the parallel time complexity for modifying a rule is $O(\max[2^s, \log N])$.

6.3 Deletion

After the RID check, suppose RID R already exists in the rule set; hence $\exists i \in \{0, 1, \ldots, N-1\}$ such that $R_i = R$. Rule deletion can be performed as: Given a RID R, delete the rule with RID R_i from the rule set. i.e., R_i should no longer produce any matching result.

To handle rule deletion, let us first consider all the n rules handled by a particular horizontal pipeline consisting of $\lceil \frac{L}{s} \rceil$ PEs. We propose to use n *valid bits* to keep track of all the n rules. A valid bit is a binary digit indicating the validity of a specific rule. A rule is valid only if its corresponding valid bit is set to "1".

For a rule to be deleted, we reset its corresponding valid bit to "0". An invalid rule is not available for producing any match result. We show an example for rule deletion in Fig. 8. In this example, initially R_0 and R_1 are valid; R_2 is invalid. R_1 is to be deleted from the rule set. During the deletion, the valid bit corresponding to R_1 is reset to "0". The n valid bits are directly ANDed with the bit vector of length n propagated through the horizontal pipeline. As a result, if a rule is invalid, the corresponding position for this

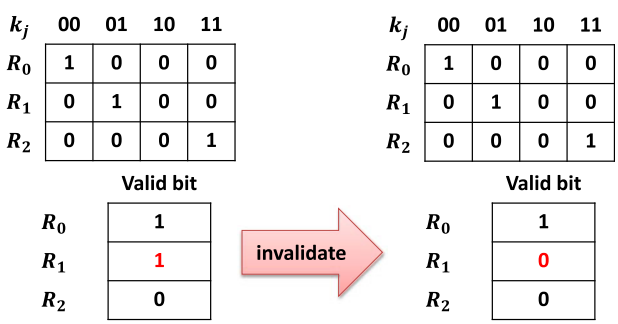

Fig. 8. Deleting an old rule R_1.

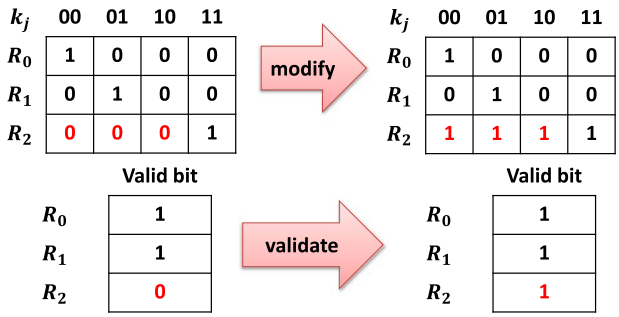

Fig. 9. Inserting a new rule R as R_2.

rule in the final AND result can only be "0", indicating the input does no match this rule.

6.4 Insertion

After the RID check, suppose RID R does not exist in the rule set; hence $\forall i \in 0, 1, \ldots, N-1$, $R_i \neq R$. Rule insertion can be performed as: Given a rule with RID R, along with all of its field values and priority, add the new rule with RID R into the rule set. i.e., check the valid bits, modify one of the invalid rules and validate this new rule.

To insert a rule, (1) we first check whether there is any invalid rule in the rule set; we denote this process as *validity check*. (2) Then we reuse the location of any invalid rule to insert the new rule: we modify one of the invalid rules into the new rule by following the same algorithm presented in Section 6.2. (3) Finally, we validate this new rule by updating its corresponding valid bit.

Fig. 9 shows an example of rule insertion in a subfield. In this figure, initially rule R_2 is invalid as indicated by the valid bit. During insertion, the location in the BV array corresponding to R_2 is reused by the new rule R. We validate the new rule R by setting its valid bit to "1".

6.5 Architecture for Dynamic Updates

The main idea of self-reconfiguration is to give each PE the ability to reconfigure its memory contents by itself during an update. In this case, the BV arrays do not have to be fed into the data memory explicitly; instead, only the rules are provided to each PE. As a result, the dynamic updates are performed in a fine-grained distributed manner in parallel; no centralized controller is required, and the amount of data I/O operations is minimized.

6.5.1 Storing Valid Bits

The data memory of the modular PE in Fig. 5 can also be used to store the valid bits. We use an extra column of PEs, each storing n valid bits for each horizontal pipeline. We place this column of PE as the first vertical pipeline on the left of the two-dimensional pipelined architecture. This is because: for each horizontal pipeline, a validity check is required for rule deletion/insertion; if the validity check is performed in the middle or at the end of the horizontal pipeline, all the packet headers being processed in the pipeline after the validity check may still use obsolete data values and produce incorrect computation results (data hazard). In that case, either stalling or flushing the pipeline is necessary, which affects the sustained throughput of the

pipeline. Storing valid bits in the first PE of each horizontal pipeline reduces the number of bubbles injected into the pipeline and minimizes the negative effect of validity check on the throughput performance.

Valid bits are extracted during run-time and output to the next PE in the horizontal pipeline. The resulting overall architecture has $\lceil \frac{N}{n} \rceil$ rows and $(\lceil \frac{L}{s} \rceil + 1)$ columns, where valid bits are stored and extracted in PE$[l, 0]$, $l = 0, 1, \ldots, \lceil \frac{N}{n} \rceil - 1$ (the first column).

6.5.2 Logic-Based Rule Decoder

To save I/O pins on FPGA, we use the pins for packet header (L pins) to input a new rule (in two cycles for ternary strings). In each PE$[l, 0]$, $l = 0, 1, \ldots, \lceil \frac{N}{n} \rceil - 1$, the RID of the rule that needs an update is provided to the rule decoders. A rule decoder is in charge of:

1) *RID check* (for all types of update operations, only in the PEs of the first column). The rule decoders check whether the RID of the new rule already exists in the rule set. This is implemented using $\log(N)$-bit comparators.
2) *Rule translation* (for modification/insertion, in all the PEs except the first column). Based on the new rule, all the data to be written (2^s bits) to the data memory are generated by the rule decoder. The logic is simple (e.g., enumerations) since s is usually small (see Section 7.3). Rule translation requires 3 clock cycles in each PE since the new rule has to be provided in 2 cycles.
3) *Validity check* (for insertion, only in the PEs of the first column). The rule decoder reports the position of the first invalid bit in the data memory. The logic for this process is also simple.
4) *Construction of valid bits* (for insertion, only in the PEs of the first column). Similar to rule translation, the rule decoder provides the up-to-date valid bits to the data memory.

Besides the four major functions mentioned above, the rule decoder is also a distributed controller for each PE. The rule decoder in a particular modular PE can also insert pipeline bubbles by resetting the output register (using the signal $reset$ in Fig. 5).

6.6 Update Schedule and Overhead

To further improve the performance of our architecture, let us investigate the update overheads in each PE for different types of update operations in Table 3.

1) For all update operations, the RID of the new rule has to be checked against all the rules, which results in one-cycle overhead.
2) The rule decoder generates control signals based on the new rule provided on the s packet header input pins/wires; hence the rule translation cannot be overlapped with the packet header matching process. This leads to a three-cycle overhead.
3) We overlap the validity check process with the on-going packet header matching process; the validity check results are reported every clock cycle to the rule decoder for all the PE$[l, 0]$, $l = 0, 1, \ldots, \lceil \frac{N}{n} \rceil - 1$.
4) To update the BV array, the rule decoder initiates 2^s memory write accesses. During memory write accesses, the memory cannot be read by the packet header matching process.
5) The priority encoders require $\log N$ cycles to modify the priority of a rule. However, the process of updating the priority can be overlapped with the process of updating the data memory.
6) The update process for the valid bits (one memory write per PE in the first column) cannot be overlapped with the on-going packet header matching process; however, it can be overlapped with the update process of the BV arrays (2^s cycles) for rule modification/insertion.

TABLE 3
Update Overhead (Clock Cycles)

Update types	Modify	Delete	Insert
RID check	1	1	1
Rule translation	3	-	3
Validity check	Overlapped with packet matching		
Updating BV array	2^s	-	2^s
Updating priority	$\log N$	$\log N$	$\log N$
Updating valid bits	-	1	1
Worst-case total overhead	$4+$ $\max[2^s, \log N]$	$1+$ $\log N$	$4+$ $\max[2^s, \log N]$

In the worst case, a single rule modification requires all the BV arrays stored in a horizontal pipeline to be updated. We show an example of the update schedule (4×3 PEs, excluding priority encoders) in Fig. 10. In this example, we assume the RID of the new rule exists in the last horizontal pipeline; this can be identified by the RID check. Therefore only the PEs in the last row require the contents in the data memories to be overwritten. As can be seen, although $(\lceil \frac{L}{s} \rceil + \lceil \frac{N}{n} \rceil)$ clock cycles are required to propagate the new rule across the entire PE array, this amount of time does not contribute to the total update overhead. This is because the update is performed in a distributed and pipelined manner. Assuming $2^s \geq \log N$, the matching process in a particular PE has to be stalled for a total number of $(2^s + 4)$ cycles for a rule modification/insertion. In Fig. 10, for instance, the matching process is stalled for $(2^s + 4) = 6$ clock cycles in each PE.

Note all other PEs except the first column neither perform RID check nor update valid bits. Also, Table 3 lists the worst-case total overhead for any PE. As can be seen, the rule insertion introduces the most overhead among all types of update operations.

7 Performance Evaluation

7.1 Experimental Setup

We conducted experiments using Xilinx ISE Design Suite 14.5, targeting the Virtex 6 XC6VLX760 FFG1760-2 FPGA [22]. This device has $118,560$ logic slices, 1200 I/O pins, 26 Mb BRAM (720 RAMB36 blocks), and can be configured to realize large amounts of distRAM (up to 8 Mb). A

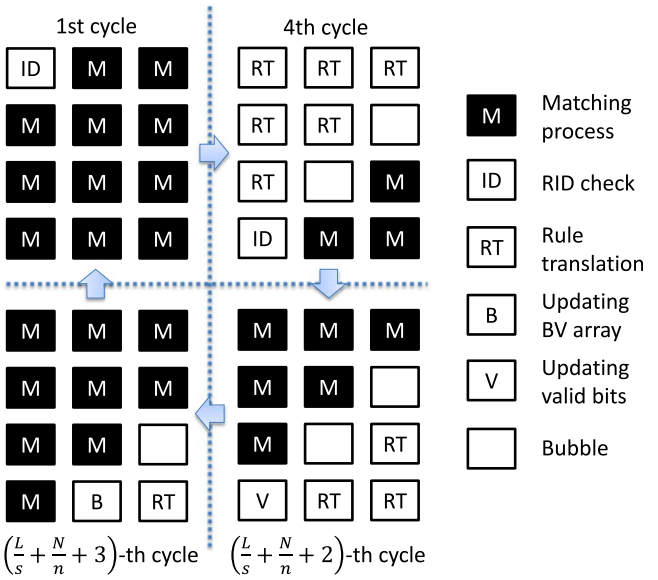

Fig. 10. Example: inserting a new rule ($\lceil\frac{L}{s}\rceil+1=3$, $\lceil\frac{N}{n}\rceil=4$, $s=n=1$).

Configurable Logic Block (CLB) on this FPGA consists of two slices, each slice having four LUTs and eight flip-flops. Clock rate and resource consumption are reported using post-place-and-route results.

Note in our approach, the construction of BVs does not explore any rule set features[6], the performance of our architecture is rule-set-independent. We use randomly generated bit vectors; we also generate random packet headers for both the classic ($d=5$, $L=104$) and OpenFlow ($d=15$, $L=356$) packet classification in order to prototype our design, although our architecture neither restricts the number of packet header fields (d) nor requires a specific length of the packet header (L). The number of rules in a rule set is chosen to be from 128 to 1 K, since most of the real-life rule sets are fairly small [8], [13]. The rest of this section is organized as follows:

- (Section 7.2) We introduce the design paramters and performance metrics.
- (Section 7.3) We optimize our design for given N and L while varying n and s.
- (Section 7.4) We scale our architecture with respect to N and L, based on the values of n and s which give the best performance in Section 7.3.
- (Section 7.5) We demonstrate our architecture supports fast dynamic updates with high sustained throughput.
- (Section 7.6) We show the latency introduced by our architecture and compare it with state-of-the-art packet classification engines.
- (Section 7.7) We report resource consumption and show the energy efficiency of our architecture under various scenarios.
- (Section 7.8) We compare the throughput, latency, and energy efficiency of our design with state-of-the-art packet classification engines.

6. e.g., the number of unique values in each field/subfield, the average length of prefixes, etc.

TABLE 4
Clock Rate (MHz) of Various Designs

		s						
		1	2	3	4	5	6	7
n	4	225.48	204.42	339.79	346.14	364.56	379.65	339.79
	8	210.08	254.97	352.86	389.86	364.30	380.47	257.47
	16	257.40	279.96	373.00	370.10	373.00	363.77	289.10
	32	259.40	239.69	342.35	344.83	355.11	315.26	262.67
	64	201.01	244.26	315.76	317.56	336.36	299.67	260.28

7.2 Design Parameters/Performance Metrics

We vary several design parameters to optimize our architecture:

- *Size of the rule set (N)*. The total number of rules
- *Length of the BV (n)*. The number of bits in the BV produced by a single PE
- *Packet header length (L)*. The total number of bits for the input packet header
- *Stride (s)*. The number of bits for a subfield
- *Update rate (U)*. The total number of all the update operations (modification, deletion or insertion) for the rule set per unit time

We study the performance trade-offs with respect to the following metrics:

- *Peak throughput (T_{peak})*. The maximum number of packets that can be processed per second without any update operation
- *Sustained throughput ($T_{sustain}$)*. The number of packets processed per second considering all update operations
- *Latency (Γ)*. The processing latency of a single packet when no dynamic update is performed
- *Resource consumption*. The total amount of hardware resources (logic slices, I/O, etc.) consumed by the architecture on FPGA
- *Energy efficiency (η)*. The total energy spent to classify an incoming packet [24]

7.3 Empirical Optimization of Parameters

To find the optimal values of n and s by experiments, we first fix the values of $N=128$, and $L=356$ for OpenFlow packet classification. The values of n and s achieving the best performance are used later for other values of N and L.

We show the maximum clock rate achieved by various designs in Table 4. We choose s from 1 to 7 and n from 4 to 64, since for $s>7$ or $n>64$, the clock rate drops to below 200 MHz. As can be seen, we achieve very high clock rate ($200\sim400$ MHz) with small variations among various designs. All the memory modules are configured to be dual-ported, hence we achieve $400\sim800$ MPPS throughput for OpenFlow packet classification. We can observe that:

1) For $s\leq2$, BV arrays are stored in 2^s-input "shallow" memories. This memory organization underutilizes the six-input LUT-based distRAM modules on FPGA. Also, since we have a large number of PEs for $s\leq2$, the entire architecture consumes large amounts of registers; the complex routing between

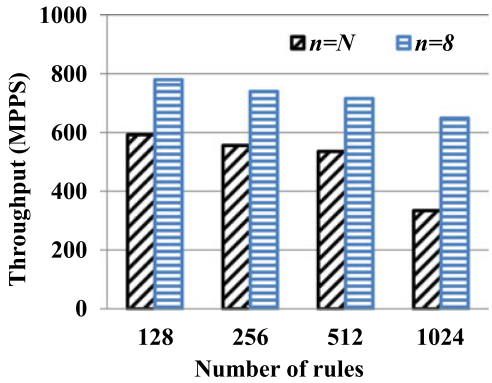

Fig. 11. Scalability with respect to N and n.

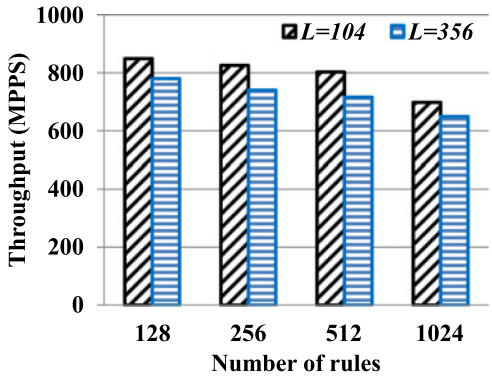

Fig. 12. Scalability with respect to N and L.

these registers also limits the maximum achievable clock rate.

2) For $3 \leq s \leq 6$, the best performance is achieved for $n = 8$ or $n = 16$. There is fast interconnect in a slice, then slightly slower interconnect between slices in a CLB, followed by the interconnect between CLBs. A PE with $n = 8$ uses exactly eight flip-flops of a slice to register a BV, while a PE with $n = 16$ uses exactly all 16 flip-flops in a CLB to register a BV. These two configurations introduce the least routing overhead.
3) For $s > 6$, BV arrays are stored in 2^s-input "deep" memories. This organization requires multiple LUTs of different CLBs to be used for a single PE; the long wiring delay between CLBs results in clock rate deterioration.
4) The performance for $s = 4$ and $n = 8$ is the best. This is because all the LUTs inside a single slice can be used as 128-bit dual-ported distRAM; the configuration of $n = 8$ and $s = 4$ not only uses up all the eight flip-flops in a slice, but also provides a memory organization to store bit vectors of total size $2^s \times n = 128$ bits.

In summary, for $N = 128$ and $L = 356$, the best performance is achieved when $s = 4$ and $n = 8$. Hence we use $s = 4$ and $n = 8$ to implement our architecture for other values of N and L[7].

7.4 Scalability of Throughput

Using $s = 4$ and $n = 8$, we vary N and L, respectively, to show the scalability of the throughput performance.

Fig. 11 shows the throughput of our architecture with respect to various values of N ($L = 356$). As can be seen, our architecture achieves very high clock rate (324 MHz) and throughput (648 MPPS) even for $N = 1{,}024$ (largest to the best of our knowledge). We also show in the same figure the necessity of using modular PEs along with the clustering technique. Compared to the basic pipelined architecture ($n = N$), our architecture achieves better throughput (up to $2\times$) when the rule set is large; in our architecture, the clock rate tapers much slower as N increases.

Fig. 12 shows the throughput for both the classic packet classification ($L = 104$) and OpenFlow packet classification ($L = 356$). Our architecture achieves very high throughput for the classic packet classification. The OpenFlow packet classification consumes more resources and requires more complex routing; hence the performance degrades compared to the classic packet classification.

7.5 Updates and Sustained Throughput

As discussed in Section 6.6, the rule insertion stalls the packet header match process for the most number of clock cycles; for the worst-case analysis, we assume pessimistically that all the update operations are rule insertions. Based on Table 3, the sustained throughput can be calculated using the following equation:

$$T_{sustain} = T_{peak} \cdot \frac{f - 2 \cdot U \cdot (4 + \max[2^s, \log N])}{f}, \quad (1)$$

where f denotes the maximum clock rate achieved for a specific design. The factor of 2 comes from the fact that memory write accesses are single-ported.

We vary the value of U and show the sustained throughput of our architecture in Fig. 13, considering the worst-case scenario for all update operations. In this implementation, $s = 4$ and $n = 8$ are used for $N = 1{,}024$. As can be seen, our architecture sustains a high throughput of 650 MPPS with 1 M updates/s, although 1 M updates/s is pessimistic considering real-world traffic.[8]

7.6 Scalability of Latency

We show the latency performance with respect to various values of N and L[9] in Fig. 14. In the same figure, we also break down the latency introduced by the two-dimensional pipelined architecture and the tree-based priority-encoders. As can be seen, more than 86 percent of the latency is introduced by the two-dimensional pipeline: $(\lceil \frac{L}{s} \rceil + \lceil \frac{N}{n} \rceil)$ cycles. The latency introduced by the priority encoders can be neglected; hence $\Gamma \sim \left(\lceil \frac{L}{s} \rceil + \lceil \frac{N}{n} \rceil\right)$. This means, for a specific configuration on s and n, and fixed values of L (or N), Γ is sublinear with respect to N (or L).

7. The choice of s and n is not unique; e.g., latency can be used as a metric to choose s and n. However, in our experiments, we choose s and n such that they give the highest clock rate, since our goal is to achieve high throughput. Other choices are also possible but they all achieve similar performance.

8. Typical update rates are ≤ 10 K updates/s [8].
9. Similar performance can be seen for other values of N, L.

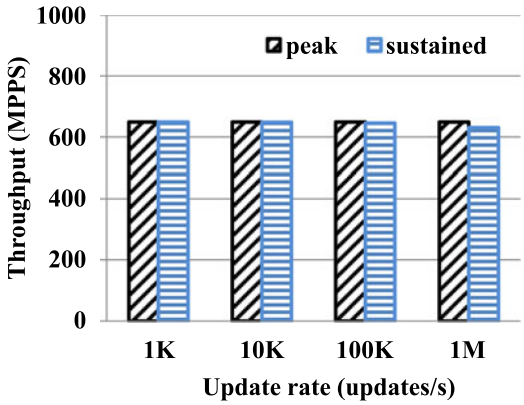

Fig. 13. Sustained throughput.

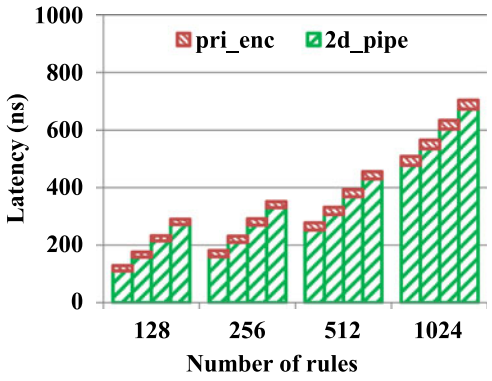

Fig. 14. Latency: for each N, the four columns correspond to $L = 89, 178, 267, 356$ from left to right.

TABLE 5
Resource Consumption ($s = 4$, $n = 8$ and $L = 356$)

No. of rules N	128	256	512	1,024
No. of logic slices	14,773	29,056	57,209	112,812
(% of total)	(12%)	(25%)	(48%)	(95%)
No. of I/O pins	722	723	724	725
(% of total)	(60%)	(60%)	(60%)	(60%)
No. of registers	48,704	97,502	195,164	329,690
(% of total)	(5%)	(10%)	(20%)	(34%)

7.7 Resource and Energy Efficiency

We report the resource consumption for OpenFlow packet classification in Table 5. The resources consumed by the architecture increases sublinearly with respect to N.

We measure the energy efficiency with respect to the energy consumed for the classification of each packet (J/packet); a small value of this metric is desirable. In Fig. 15, we show the energy efficiency without the power gating technique (w/o opt.) and with the power gating technique (w/ opt.) as discussed in Section 5, respectively. Three scenarios for input packet headers are tested with 1 K OpenFlow rule set, including:

1) *All-match*. Every input packet header produces an "all-one" BV in any PE. This means every input packet header matches all the rules, which is too pessimistic.
2) *Random*. Packet headers are generated randomly.

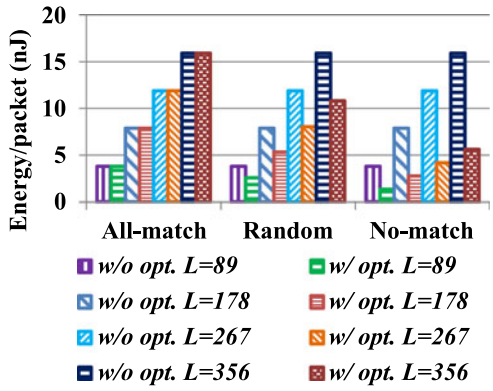

Fig. 15. Energy efficiency ($s = 4$, $n = 8$, $N = 1{,}024$).

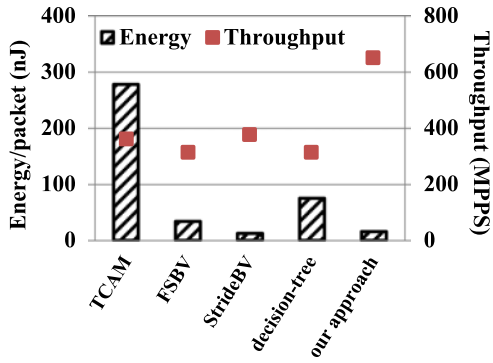

Fig. 16. Comparing throughput and energy efficiency.

3) *No-match*. Every input packet header is identified as not matching any of the 1 K rules in the first column of PEs. In this case, since the memory modules in all the other columns of PEs are deactivated, the energy efficiency of our design with power gating technique is optimistic.

For each scenario, we vary the number of horizontal pipeline stages to investigate the energy efficiency. The power gating technique is more effective for larger two-dimensional pipelined architectures[10]; this is because more data memory ports can be turned off if an early stage (close to the first column of PEs) reports no match. As can be seen in Fig. 15, with the power gating technique, our design can save up to 67 percent energy; the actual energy saved depends on the pattern of the input packet headers.

7.8 Comparison with State-of-the-art

7.8.1 Throughput and Energy Efficiency

Fig. 16 shows a comparison of our approach with existing hardware accelerators. We consider 15-field OpenFlow classification rule sets for all the schemes. To make a fair comparison, all the implementations of TCAM [9], FSBV [21], StrideBV [13], decision-tree on FPGA [10], and our approach support 1 K rules.

We scale the TCAM performance to the state-of-the-art technology based on a 18 Mbit TCAM running at 360 MHz and consuming 15 W [9]; we ignore the power consumed by the extra logic for managing the TCAM access. We linearly scale up the memory consumption of FSBV (29 Bytes per

10. The energy saving is also remarkable as we scale up N.

104-bit rule) and StrideBV (156 Bytes per 104-bit rule) to estimate the total power consumed for 1 K 356-bit rules; the power consumption for FPGA-based implementations is evaluated using XPower Analyzer tool available in the Xilinx ISE Design Suite 14.5 targeting Virtex 6 XC6VLX760 FFG1760-2 FPGA. The resource consumption of the decision-tree-based approach on FPGA [10] is also scaled to the same Virtex 6 device (10,307 logic slices, 223 I/O pins, and 407 BRAM modules). For StrideBV, the most energy-efficient design with $s = 4$ is considered. The energy consumption of our approach is based on the design with $s = 4$ and $n = 8$, considering the worst-case "All-match" scenario as discussed in Section 7.7.

Since our design runs at a higher clock rate than other FPGA-based designs (FSBV, StrideBV, and decision-tree-based approaches), to make a fair comparison, we also show the throughputs of all the approaches for OpenFlow packet classification in Fig. 16. The throughput performance of TCAM is estimated based on the assumption that packets can be classified within a single clock cycle. For the FSBV and StrideBV approaches, we measure the throughput for the single-pipeline implementation due to limited memory resources on FPGA. We observe:

- All the FPGA-based approaches achieve at least 4× energy efficiency than the TCAM solution.
- Compared to FSBV (33.8 nJ/packet), our approach (15.9 nJ/packet) consumes less energy to classify an OpenFlow packet. The longest wire length is reduced in our architecture, leading to a more energy-efficient design on FPGA.
- Compared to the decision-tree-based approach on FPGA (75.1 nJ/packet), our approach achieves 5× energy efficiency; our approach only uses LUT-based distRAM, while the decision-tree-based approach employs a large amount of BRAM and distRAM at the same time. Note that the throughput performance of decision-tree-based approach depends on the rule set.
- Compared to StrideBV (12.7 nJ/packet), our approach consumes more energy (1.25×). Note the PE in our architecture is self-reconfigurable and supports dynamic updates, which requires more resources and consumes more energy. Moreover, our architecture allows us to classify packets at a very high clock rate, which also results in more power consumption than other approaches. However, with slightly more energy, our approach achieves scalability, sustains high throughput (2×) and supports fast incremental update.

7.8.2 Throughput and Latency

We also compare the latency performance for various approaches in Fig. 17. We assume the TCAM can classify packets in a single clock cycle [9]. We assume seven pipeline stage are employed in the FSBV-based approach, while the clock rate can be sustained at 167 MHz [21]. For StrideBV [13], the latency estimation is based on a single pipeline consisting of $\lceil \frac{L}{s} \rceil = 89$ stages running at 105 MHz. We assume the decision-tree-based implementation [10] employs a 16-stage pipeline clocked at 125 MHz.

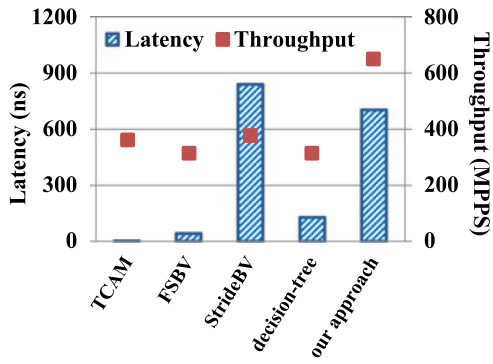

Fig. 17. Comparing throughput and latency.

As can be seen, the TCAM has the lowest latency (single-cycle classification). The StrideBV and our approach introduce the highest latency due to the deeply pipelined architecture. However, our approach eliminates long wires and increases the clock rate; this in turn improves the latency performance (703 ns) compared to StrideBV (839 ns).

8 Conclusion

In this paper we presented a two-dimensional pipelined architecture for packet classification. The advantages of the proposed architecture include:

1) *Parameterized*. The architecture is highly parameterized; it can be optimized with respect to various performance metrics.
2) *Rule-set-independent*. The performance does not depend on any rule set features other than the rule set size.
3) *High-throughput*. All the PEs access their designated distRAM modules independently. The memory access is localized, resulting in shorter interconnections in each PE. This leads to high clock rate and high throughput on FPGA.
4) *Scalable with respect to rule set size*. The longest wire length is not significantly affected by the total number of rules; the architecture sustains high throughput for a large number of rules, assuming we have sufficient hardware resources.
5) *Scalable with respect to input length*. The throughput is not adversely affected by the length of the packet header. Our architecture achieves good performance for both classic and OpenFlow packet classification.
6) *Dynamically updatable*. The dynamic update is performed in a distributed manner on self-reconfigurable PEs; the update scheme has little impact on the sustained performance.
7) *Energy-efficient*. The proposed architecture demonstrates better energy efficiency. Compared to StrideBV, our approach sustains 2× throughput and supports fast dynamic updates with slightly more energy consumption.

In the future, we plan to use this architecture vigorously for other network applications including traffic classification and heavy hitter detection for data center networks. We will also explore more techniques to improve the energy efficiency of this architecture.

ACKNOWLEDGMENTS

This work is supported by the US National Science Foundation (NSF) under grant No. CCF-1320211. Equipment grant from Xilinx is gratefully acknowledged. Y. R. Qu is the corresponding author.

REFERENCES

[1] OpenFlow Switch Specification V1.3.1. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf, 2012.
[2] Y. R. Qu, S. Zhou, and V. K. Prasanna, "High-performance architecture for dynamically updatable packet classification on FPGA," in *Proc. ACM/IEEE Symp. Arch. Netw. Commun. Syst.*, 2013, pp. 125–136.
[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.
[4] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Netw.*, vol. 15, no. 2, pp. 24–32, Mar. 2001.
[5] P. Gupta and N. McKeown, "Dynamic algorithms with worst-case performance for packet classification," in *Proc. Eur. Commission Int. Conf.*, 2000, pp. 528–539.
[6] 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.
[7] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 193–204, 2005.
[8] B. Vamanan and T. N. Vijaykumar, "TreeCAM: Decoupling updates and lookups in packet classification," in *Proc. Conf. Emerging Netw. Exp. Technol.*, 2011, pp. 27:1–27:12.
[9] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-efficient TCAMs for forwarding engines," in *Proc. Joint Conf. IEEE Comput. Commun.*, vol. 1, 2003, pp. 42–52.
[10] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Trans. VLSI Syst.*, vol. 20, no. 9, pp. 1668–1680, Sep. 2012.
[11] J. Bispo, I. Sourdis, J. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," in *Proc. IEEE Int. Conf. Field Programmable Technol.*, 2006, pp. 119–126.
[12] Z. P. Ang, A. Kumar, and Y. Ha, "High speed video processing using fine-grained processing on FPGA platform," in *Proc. IEEE Int. Symp. Field-Programmable Custom Comput. Mach.*, 2013, pp. 85–88.
[13] T. Ganegedara and V. K. Prasanna, "StrideBV: Single chip 400g+ packet classification," in *Proc. IEEE Int. Conf. High Perform. Switching Routing*, 2012, pp. 1–6.
[14] I. Bonesana, M. Paolieri, and M. Santambrogio, "An adaptable FPGA-based system for regular expression matching," in *Proc. Des., Autom. Test Eur.*, 2008, pp. 1262–1267.
[15] A. Sudarsanam, R. Barnes, J. Carver, R. Kallam, and A. Dasu, "Dynamically reconfigurable systolic array accelerators: A case study with extended Kalman filter and discrete wavelet transform algorithms," *Comput. Digit. Techn., IET*, vol. 4, no. 2, pp. 126–142, 2010.
[16] R. Salvador, A. Otero, J. Mora, E. de la Torre, T. Riesgo, and L. Sekanina, "Self-reconfigurable evolvable hardware system for adaptive image processing," *IEEE Trans. Comput.*, vol. 62, no. 8, pp. 1481–1493, Aug. 2013.
[17] L. Frigerio, K. Marks, and A. Krikelis, "Timed coloured petri nets for performance evaluation of DSP applications: The 3GPP LTE case study," in *Proc. VLSI-SoC: Des. Methodol. SoC and SiP*, 2010, vol. 313, pp. 114–132.
[18] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, Jan./Feb. 2000.
[19] Y. R. Qu, S. Zhou, and V. Prasanna, "Scalable many-field packet classification on multi-core processors," in *Proc. Int. Symp. Comput. Archit. High Perform. Comput.*, 2013, pp. 33–40.
[20] V. Pus and J. Korenek, "Fast and scalable packet classification using perfect hash functions," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2009, pp. 229–236.
[21] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using FPGAs," in *Proc. Annu. Symp. Parallelism Algss. and Archit.*, 2009, pp. 188–196.
[22] Virtex-6 FPGA Family. [Online]. Available: http://www.xilinx.com/products/virtex6, 2012.
[23] Y.-H. E. Yang and V. K. Prasanna, "High throughput and large capacity pipelined dynamic search tree on FPGA," in *Proc. 18th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2010, pp. 83–92.
[24] A. Kennedy, X. Wang, and B. Liu, "Energy efficient packet classification hardware accelerator," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2008, pp. 1–8.

Yun R. Qu received the BS degree in electrical engineering from Shanghai Jiao Tong University, and the MS degree in electrical engineering at the University of Southern California, in 2009 and 2011, respectively. He is currently working towards the PhD degree in computer engineering at the University of Southern California. His research interests include large-scale regular expression matching, IP address lookup, multi/many-field packet classifier, and online traffic classification engine for network routers. He has also done research in error correcting codes and communication theory. His primary focuses are on algorithm design, algorithm mapping onto custom hardware, high-performance and power-efficient architectures. He is a member of the IEEE.

Viktor K. Prasanna received the BS degree in electronics engineering from the Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from the Pennsylvania State University. He is a Charles Lee Powell chair in engineering in the Ming Hsieh Department of Electrical Engineering and a professor of computer science at the University of Southern California (USC). His research interests include high-performance computing, parallel and distributed systems, reconfigurable computing, and embedded systems. He is the executive director of the USC-Infosys Center for Advanced Software Technologies (CAST) and is an associate director of the USCChevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). He also serves as the director of the Center for Energy Informatics at USC. He served as the editor-in-chief of the *IEEE Transactions on Computers* during 2003-06. Currently, he is the editor-in-chief of the *Journal of Parallel and Distributed Computing*. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering co-chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the steering chair of the IEEE International Conference on High Performance Computing (HiPC). He is the recipient of the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University. He is a fellow of the IEEE, the ACM and the American Association for Advancement of Science (AAAS).