High-performance Architecture for Dynamically Updatable Packet Classification on FPGA*

Yun R. Qu, Shijie Zhou, and Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Engineering, University of Southern California
Los Angeles, CA 90089, U.S.A
yunqu@usc.edu, shijiezh@usc.edu, prasanna@usc.edu

ABSTRACT

Algorithms and FPGA based implementations for packet classification have been studied over the past decade. Algorithmic solutions have focused on high throughput; however, supporting dynamic updates has been challenging. In this paper, we present a 2-dimensional pipelined architecture for packet classification on FPGA, which achieves high throughput while supporting dynamic updates. Fine grained processing elements are arranged in a 2-dimensional array; each processing element accesses its designated memory locally, resulting in a scalable architecture. The entire array is both horizontally and vertically pipelined. As a result, it supports high clock rate that does not deteriorate as the length of the packet header or the size of the rule set increases. The performance of the architecture does not depend on rule set features such as the number of unique values in each field. The architecture also efficiently supports range searches in individual fields. The total memory is proportional to the rule set size. Dynamic updates– modify, delete and insert operations for the rule set during run-time are also supported on the self-reconfigurable processing elements with very little impact on the sustained throughput. Experimental results show that, for a 1K 15-tuple rule set, a state-of-the-art FPGA can sustain 190 Gbps throughput with 1 million updates/second. To the best of our knowledge, we are not aware of any packet classification approach that simultaneously supports both high throughput and dynamic updates of the rule set. Our architecture demonstrates 4× energy efficiency while achieving 2× throughput compared to TCAM.

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*, and uses a flexible protocol– OpenFlow [1] to manage network traffic. One of the kernel function OpenFlow performs is the *flow table lookup* [2, 3]. The flow table lookup requires multiple fields of the incoming packet to be matched against entries in a prioritized flow table. This is similar to the classic multi-field packet classification mechanism [4].

Many existing solutions for multi-field packet classification employ ternary content addressable memories (TCAMs) [5, 6]. TCAMs cannot support efficient dynamic updates; they

are expensive and power-hungry. TCAM-based solutions also suffer from range expansion when converting ranges into prefixes [6].

Field-Programmable Gate Array (FPGA) technology has been used to implement algorithmic solutions for real-time network processing [7, 8]. FPGA-based packet classification engine can achieve very high throughput for rule sets of moderate size [9]. However, as the number of packet header fields or the rule set size increases, FPGA-based approaches often suffer from clock rate degradation. Since the OpenFlow protocol requires a large number of packet header fields to be examined [2], OpenFlow packet classification remains a challenging research problem.

Future network applications following OpenFlow standard require the hardware to perform frequent incremental updates during run-time. Since it is prohibitively expensive to reconstruct an optimal architecture repeatedly for timely updates, a flexible and run-time reconfigurable hardware-based engine is needed for OpenFlow packet classification. While many sophisticated solutions have been proposed for packet classification supporting dynamic updates over the years [10], due to the rapid growth of the network size and the bandwidth requirement of the Internet, supporting dynamic update without loss of throughput performance remains challenging.

With the above challenges in mind, 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 and 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. Specifically, our contributions in this work include:

- *Scalable architecture*: A 2-dimensional pipelined architecture on FPGA, which achieves scalability with respect to the size of the rule set and sustains high throughput for OpenFlow packet classification.

- *Distributed update algorithms*: A set of algorithms supporting dynamic updates– modify, delete and insert operations for rule set on the proposed architecture. The algorithms are performed distributively on self-reconfigurable PEs. The update operations have little impact on the sustained throughput performance.

- *Implementation tradeoffs*: Tradeoffs between various design parameters and performance metrics, including

Table 1: Example OpenFlow packet classification rule set

RID	Ingr	Meta-data	Eth src	Eth dst	Eth type	VID	VLAN priority	MPLS label	MPLS tfc	SA	DA	Ptrl	Tos	SP	DP	Action
R_0	5	*	00:13	00:06	0x0800	*	5	0	*	001*	*	TCP	0	*	*	Action 0
R_1	*	*	00:07	00:FF	*	100	7	16000	0	00*	1011*	UDP	*	*	*	Action 1
R_2	*	*	*	00:00	0x8100	4095	7	*	*	1*	1011*	*	*	2	5	Action 0
R_3	1	*	00:FF	*	*	4095	*	*	*	1*	1*	*	0	7	5	Action 2

rule set size, throughput, update rate and resource consumption.

- *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 190 Gbps with 1 million updates/second (M updates/s) for a 1K 15-tuple rule set.

- *Energy efficiency*: Detailed comparison of our architecture with existing solutions for packet classification with respect to energy efficiency. Compared to TCAM, our architecture sustains 2× throughput and supports fast dynamic updates with 4× 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 summarize existing packet classification techniques in Section 3. We detail the design of the 2-dimensional pipelined architecture in Section 4, and we present the update algorithms on this architecture in Section 5. Section 6 provides the experimental results. Section 7 concludes the paper.

2. BACKGROUND

2.1 Classic Packet Classification

The classic packet classification [4] involves classifying packets based on the five fields in the packet header: the source/destination IP addresses, source/destination port numbers, and transport layer protocol. The individual predefined entries for classifying a packet are called *rules*; each rule is associated with a unique rule ID (RID). The data set containing all the rules is called *rule set*. Different fields in a rule require different types of match criteria such as *prefix match*, *range match*, and *exact match*. A packet is considered matching a rule only if it matches all the fields in that rule. A packet may match multiple rules, but only the rule with the highest priority is used to take action. The total number of rules in a rule set is usually less than 100K [11].

2.2 OpenFlow Packet Classification

A newer version of packet classification– flow table lookup in OpenFlow requires a larger number of packet header fields to be examined; we use *OpenFlow packet classification* and flow table lookup interchangeably in this paper. For example, in the current specification of OpenFlow protocol [2, 3], 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. We show an example rule set in Table 1. As can be seen, a specific field of each entry in the rule set for OpenFlow packet classification usually contains a prefix[1].

---

[1]A value and *ANY* (denoted as * in Table 1, which matches any value) can both be viewed as a prefix.

Table 2: Classic and OpenFlow packet classification

Type	Classic	OpenFlow
Match criteria	prefix, range, exact	prefix
Number of fields (d)	5	15
Packet header bits (L)	104 [12]	356 [2]
Update	Relatively static	$\leq$ 10K updates/s [11]

We summarize the differences between the classic packet classification and OpenFlow packet classification in Table 2. Compared to the classic packet classification, there are two new challenges for OpenFlow:

- OpenFlow requires a large number of bits and packet header fields to be processed for each packet; it is challenging to achieve high performance.

- OpenFlow places high emphasis on dynamic updates (rule modification, deletion and insertion); it is challenging to design a flexible fast update scheme for the rule set.

3. PRIOR WORK

3.1 Packet Classification Techniques

Packet classification has been extensively studied over the past decade [4, 13]. Hardware-based packet classification approaches can be categorized into two major groups: decision-tree based and decomposition based approaches.

Decision-tree based approaches involve cutting the search space into smaller subspaces based on the information from one or more fields in the rule. [14] proposes to map the decision tree onto a pipelined architecture on FPGA; for a rule set containing 10K rules, a throughput of 80 Gbps is achieved for packets of minimum size (40 bytes). However, as the number of rules N increases, the memory consumption of each pipeline stage grows at a rate of $O(N)$. As a result, more memory modules have to be used and longer wires are needed for interconnection, which adversely affects the overall pipeline throughput.

Decomposition based approaches [15, 16] first search each packet header field individually. The partial results are then merged to produce the final result. As a decomposition based approach, Bit Vector (BV) approach [17] 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 "1" only if the input matches the corresponding rule in this field. A bit-wise logical AND operation gathers the matches from all fields in parallel.

3.2 BV-based Approaches on FPGA

In [12], a Field-Split BV (FSBV) approach on FPGA was proposed. In the FSBV approach, each rule in a W-bit field is represented by a W-bit ternary string; each string consists
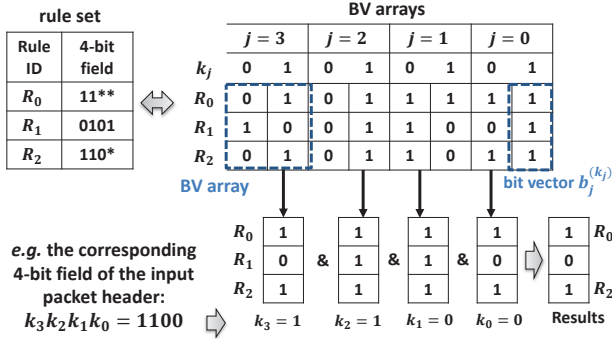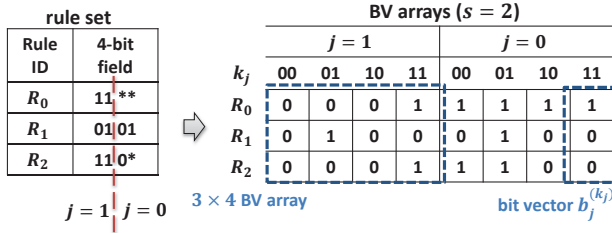
Figure 1: BV approach ($L = 4$)

Figure 2: Striding

Figure 3: Basic architecture for BV-based approaches [9]

---

of W "subfields[2]" on $\{0, 1, *\}$. W-sets of bit vectors each of length N can be generated; each bit of a specific bit vector corresponds to a packet classification rule. The example in Figure 1 shows, for a rule set consisting of 3 4-bit rules ($N = 3$, $W = 4$), 4 sets of bit vectors can be generated, each of length 3. For each 1-bit subfield, a set of 2 bit vectors of length $N = 3$ are generated. A bit in the bit vectors is set to "1" only if the input matches the corresponding rule. The input bits are used directly to address the memory and extract bit vectors. FSBV proposes to perform logical AND operations in a pipelined manner on all the extracted bit vectors to generate the final match result on FPGA. [12] provides the algorithms to construct the bit vectors for prefix match or exact match fields; however, since FSBV requires rules to be represented in ternary strings, it suffers from range expansion when converting ranges into prefixes [18].

In the FSBV approach, the memory organization for each pair of bit vectors (2 addresses, N-bit data) underutilizes the memory on FPGA (See Section 6), which affects the achievable clock rate negatively. One optimization for this is to use striding technique– StrideBV [9]. The basic idea is to partition each field into many multi-bit subfields (strides), instead of single-bit subfields in the FSBV approach. As shown in Figure 2, to map the StrideBV approach onto FPGA, a memory with 4 addresses and data width of 3 bits is needed in each pipeline stage. However, StrideBV still does not support range match for packet classification. Also, as the number of rules N increases, the clock rate often deteriorates since wires of length $O(N)$ have to be used for the memory storing N-bit data.

3.3 Supporting Dynamic Updates

Although dynamic updatable packet classification has been a well-defined problem for years [10], we are not aware of any

---

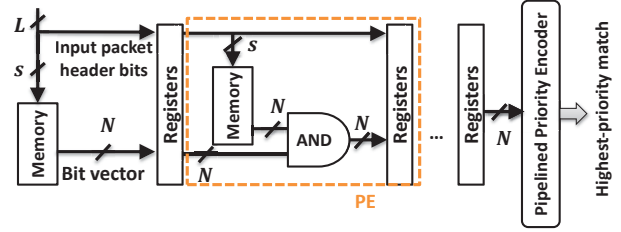[2]Defined as a specific bit position of the classification rules.

---

solution on FPGA which supports both high performance and fast dynamic updates. [10] proposes two algorithms based on tree/trie structures to support dynamic updates. These algorithms 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 most decision-tree based approaches, if the rule set needs to be updated, the tree must be recomputed and mapped onto FPGA, which is very expensive. Some of the decomposition based approaches [16] explore the use of external memory; for each update, a number of external memory write accesses must be performed. This is also very expensive. In this paper, we present a high-performance architecture for packet classification; each PE in this architecture is self-reconfigurable and dynamically updatable. We propose an efficient update scheme on this architecture.

4. ARCHITECTURE

4.1 Challenges

Let us denote the input bits in subfield j as k_j; we use k_j to directly index the bit vectors for subfield j. We use $b_j^{(k_j)}$ to denote the k_j-th bit vector in subfield j. Note that the length of each bit vector is N. We denote the data structure that stores the bit vectors corresponding to a given subfield as *bit vector array* (BV array), as shown in Figure 1; a BV array consists of all $b_j^{(k_j)}$ with the same j. In Figure 2, each stride of 2-bit length is associated with a $N \times 2^s = 3 \times 4$ BV array, where the index k_j for each array consists of 2 bits. An $N \times 2^s$ BV array is stored in a $2^s \times N$ memory.

We denote the length of a subfield (stride length) as s and the total length of the input packet header bits as L (*e.g.* $L = 356$ bits for OpenFlow packet classification). FSBV can be visualized as a special case where $s = 1$; we apply the FSBV approach to all the fields (in total L bits) instead of two W-bit fields [12]. Since we partition all the fields into subfields of the same length s, there are in total $\left\lceil \frac{L}{s} \right\rceil$ subfields for an input header of L bits. Thus we have all the s-bit subfields indexed as $j = 0, 1, ..., \left\lceil \frac{L}{s} \right\rceil - 1$. Figure 2 shows an example of constructing strided bit vectors ($s = 2$) from a rule set.

After we have constructed all bit vectors in all the subfields, we use the input header bits k_j's to address the corresponding bit vectors in the BV arrays. For a subfield j, $b_j^{(k_j)}$ is extracted for the input bits k_j. For example, in Figure 2, if the input has $k_0 = 10$ in the subfield $j = 0$, we have the bit vector to be extracted from memory is $b_0^{(10)} = 100$, indicating only R_0 matches the input in this subfield.

In the pipelined architecture for BV-based approaches on FPGA as shown in Figure 3 [9], each stage extracts a bit
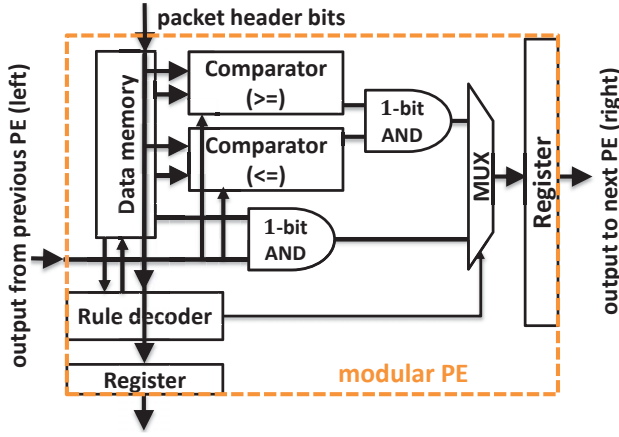
Figure 4: Modular PE supporting prefix/range match

vector for a subfield; this bit vector is ANDed with the bit vector output by the previous stage. The AND result in a particular stage is output to the next stage. Excluding the priority encoder, we have $\left\lceil \frac{L}{s} \right\rceil$ PEs[3] in this pipeline. We denote such an architecture as a *basic pipelined architecture* or *basic pipeline*. There are two main problems in this architecture:

1. The classic packet classification requires range match to be performed in the port number fields– 16-bit source port number (SP) or destination port number (DP). Since FSBV as well as StrideBV does not support range match directly, they often need a range-to-prefix conversion; this can lead to rule set expansion [18].

2. In BV-based approaches, no matter whether we use striding or not, the bit vectors in each pipeline stage for N rules are N-bit long. For distributed RAM (distRAM) or Block RAM (BRAM) module of fixed size[4], the number of memory modules needed for each pipeline stage grows linearly with respect to N. This means the length of the longest wire connecting different memory modules together also increases at a rate of $O(N)$, which degrades the throughput performance of the pipeline.

To address the first problem, we construct a modular PE which handles both range match and prefix match (Section 4.2). To address the second problem, we propose a 2-dimensional pipelined architecture. In Section 4.3, we construct horizontal and vertical pipelines using multiple modular PEs, and then present the overall architecture. Finally we show optimizations of this architecture in Section 4.4.

4.2 Modular PE

Let us first consider the internal organization of a basic PE, which handles only one rule ($N = 1$) in a 1-bit subfield ($s = 1$). We construct a *modular PE* as shown in Figure 4; such a modular PE can handle both range match and prefix match. A 2-dimensional pipelined architecture can be constructed using multiple modular PEs (Section 4.3); with

---

[3]We use PE to denote a pipeline stage that is responsible for producing a bit vector.
[4]*e.g.* 64-bit distRAM based on 6-input Look Up Table (LUT) and 36Kb SRAM-based BRAM [19].

optimization techniques, a modular PE can also handle multiple rules and strided subfields (Section 4.4).

We denote a rule requiring prefix match as a *prefix rule*, while we denote a rule requiring range match as a *range rule*. For a 1-bit subfield, the prefix rule can be handled efficiently using FSBV approach. In Figure 4, the packet header bit is used to address the memory directly; the extracted bit vector is then ANDed with the bit vector output from the previous PE in the pipeline.

A range rule in a 1-bit subfield $[x_1, x_2]$ ($x_1 \leq x_2$) can be represented using a lowerbound x_1 and an upperbound x_2. Hence we use two parallel comparators to examine whether the input matches the range rule. We use 1 bit to indicate the result of each comparison ("0" for mismatch and "1" for match); as shown in Figure 4, a 1-bit logical AND gate is used to merge the 2 comparison results. In our implementation, the higher-order bits in a field requiring range match are always handled first in the pipeline, followed by the lower-order bits of this field. Notice that:

- The result of each comparison also depends on the comparison result from the previous modular PE (on the left, handling a higher-order bit of the range). If the previous modular PE already reports a mismatch, the output of the modular PE can only be a mismatch, since the input packet header does not match the range rule in higher-order bits.

- Suppose the previous modular PE reports a match. The "comparator ($>=$)" outputs a "1" only if the input is greater than or equal to x_1, while the "comparator ($<=$)" outputs a "1" only if the input is less than or equal to x_2.

- A multiplexer (MUX) is added into the modular PE, which is controlled by the rule decoder. Depending on the control signal that rule decoder provides, the MUX either outputs the range match result, or the prefix match result. Details about the rule decoder will be introduced later.

The amount of memory required for storing an $N \times 2^s = 1 \times 2$ BV array is 2×1 bits, while the memory required for storing a 1-bit lowerbound and a 1-bit upperbound is also 2×1 bits. This means the *data memory* shown in Figure 4 needs to be configured to have 2 locations, each storing a 1-bit data. The data memory is used either as the storage for the BV array, or as the storage for the range boundaries.

The modular PE also has other components added, such as rule decoder and the register for input packet header bits. We denote the register for the input packet header bits as the *input register*; we denote the register after MUX as the *output register*. The rule decoder is mainly used for dynamic updates (Section 5), while the input register is used to construct a vertical pipeline (Section 4.3).

4.3 2-dimensional Pipelined Architecture

To handle a larger number of rules and more input packet header bits, we use multiple modular PEs to construct a complete 2-dimensional pipelined architecture as shown in Figure 5. We define the following directions:

Horizontal the forward (right) and backward (left) direction in which the bit vectors are propagated in a pipelined fashion
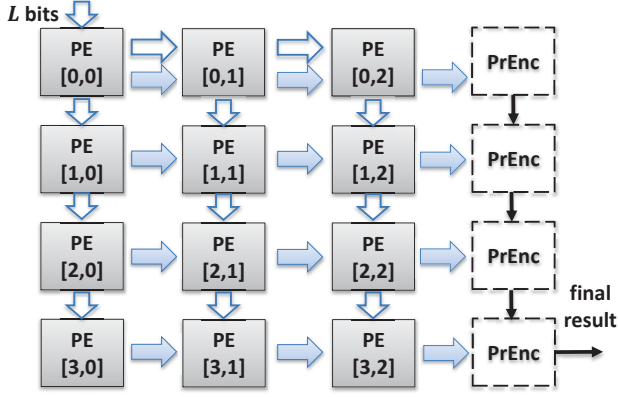
Figure 5: An example of the 2-dimensional pipelined architecture ($N = 4$, $L = 3$, 12 modular PEs)

Vertical the upward (up) and downward (down) direction in which the input packet header bits in a subfield j is propagated in a pipelined fashion

In Figure 5, we use PE$[l, j]$ to denote the modular PE located in the l-th row and j-th column, where $l = 0, 1, 2, 3$ and $j = 0, 1, 2$. We use distRAM for the memory in each PE, so that the overall architecture can be easily fit on FPGA and the memory access in each PE is localized.

For the architecture in Figure 5, we use input registers in modular PEs to construct vertical pipelines (*e.g.* PE$[0, 0]$, PE$[1, 0]$, PE$[2, 0]$, PE$[3, 0]$), while we use output registers in modular PEs to construct horizontal pipelines (*e.g.* PE$[0, 0]$, PE$[0, 1]$, PE$[0, 2]$). Notice:

1. L bits of the packet header is input from the top, and the output is finally collected in the bottom-right corner.

2. Since we do not require the rules to be arranged in the rule set following any specific order, we need a priority encoder (PrEnc) [9] at the end of each horizontal pipeline to report the highest-priority match.

3. The match results of all the horizontal pipelines are also collected by priority encoders in a pipelined fashion (a vertical pipeline for priority encoders).

4. The data propagated in the horizontal pipelines consist of bit vectors (wide shaded arrows in Figure 5); the data propagated in the first horizontal pipeline also consist of input packet header bits (wide empty arrows in Figure 5).

5. The data propagated in the vertical pipelines of PEs consist of packet header bits; the data passed between priority encoders are RIDs (narrow arrows in Figure 5).

Since each modular PE constructed in Section 4.2 performs a range/prefix match for one rule in a 1-bit subfield, the architecture in Figure 5 consisting of 4 rows and 3 columns of modular PEs can handle 4 rules, each rule having 3 1-bit subfields. Using more modular PEs, this architecture can be scaled for a large number of rules, and for long packet headers. For a rule set consisting of N rules, and an L-bit packet header, the architecture requires N rows and L columns of PEs to be connected in a pipelined fashion.
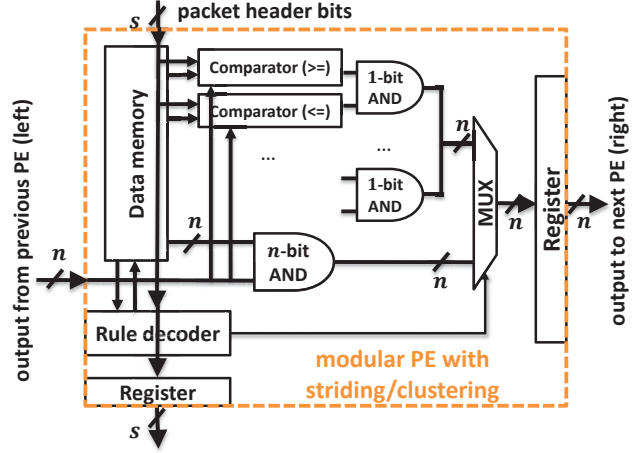

Figure 6: PE with striding and clustering techniques

4.4 Striding and Clustering

Striding technique can be applied to the modular PE shown in Figure 4. Suppose the modular PE only needs to perform packet header match against one rule. Based on the striding technique discussed in Section 3.2, the amount of memory required for prefix match in a s-bit subfield is $2^s \times 1$; for range match, the amount of memory for storing 2 s-bit range boundaries is $2 \times s$. Hence we configure the data memory in a PE to be $2^s \times s$, so that the data memory can hold either a BV array or 2 range boundaries. Also in Figure 4, the 1-bit AND gate on the right of the 2 comparators needs to be adjusted into an s-bit logical AND gate. The length of the input register is s bits because we have s input packet header bits when using striding technique.

Besides the striding technique, we also introduce a *clustering* technique for the modular PE in the architecture. The basic idea is to build a PE which can handle multiple rules instead of a single rule. Suppose the modular PE now needs to perform packet header match against n rules. We have:

1. For prefix match, 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 below the 2 comparators in Figure 4 needs to be adjusted into an n-bit logical AND gate.

2. For range match, the data memory needs to store 2 s-bit range boundaries for each rule. Therefore the data memory has to be configured as $2n \times s$. The 2 comparators in Figure 4 needs to be expanded into $2n$ comparators, and the AND gate on the right of the comparators is replaced by n parallel 1-bit AND gates.

3. To use the data memory as storage for both the BV array and the range boundaries, the data memory is configured to be $\max[2^s, 2n] \times \max[n, s]$.

We show a modular PE with striding (varying s) and clustering (varying n) techniques in Figure 6. A modular PE with striding and clustering techniques perform packet header match in an s-bit subfield against a set of n rules; the resulting 2-dimensional pipelined architecture has $\lceil \frac{N}{s} \rceil$ rows and $\lceil \frac{L}{s} \rceil$ columns of PEs. As can be seen, the modular PE without striding and clustering techniques in Figure 4 is a special case when $s = 1$ and $n = 1$.

5. DYNAMIC UPDATES

5.1 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.* bit vector, BV array, range boundary, and valid bit) that needs to be updated

Up-to-date (data structure) (data structure) that is already updated

Given a rule set Φ consisting of N rules $\{R_i | i = 0, 1, ..., N - 1\}$, 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 $i \in \{0, 1, ..., N - 1\}$ 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 $i \in \{0, 1, ..., N - 1\}$ 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 $R_i \neq R$ for $\forall i \in \{0, 1, ..., N - 1\}$[5], insert rule R into Φ (e.g. add a brand new rule as R_4 into Table 1)

Suppose in the architecture, all modular PEs are implemented with striding and clustering techniques; each of these PEs stores either (1) a BV array of size $n \times 2^s$ for a prefix match subfield, or (2) $2n$ s-bit range boundaries for a range match subfield, as discussed in Section 4.4.

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. The effect of this process is discussed in Section 5.6.

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

1. (Section 5.2) For rule modification, we update all the corresponding bit vectors in the BV arrays (for prefix match fields), or we update the range boundaries directly (for range match fields), or we update priority encoders (for priority).

2. (Section 5.3) For rule deletion, we keep a "valid" bit for each rule; we reset the bit to invalidate a rule.

---
[5]If $\exists i \in \{0, 1, ..., N - 1\}$, such that $R_i = R$, modify rule R_i.

3. (Section 5.4) For rule insertion, we check the valid bits of all 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 5.5 covers the architectural design for the update mechanism. The resulting overall architecture consists of multiple self-reconfigurable PEs; each PE configures its memory contents in a distributed manner. Section 5.6 summarizes the update schedule.

5.2 Modification

After the RID check, rule modification can be performed as: Given a rule with RID R ($\exists i \in \{0, 1, ..., N - 1\}$ such that $R_i = R$, i.e., RID R already exists in the rule set), all of its field values and priority, (1) compute the up-to-date bit vectors, and replace the outdated bit vectors in the BV arrays with the up-to-date bit vectors; or (2) update the range boundaries of rule R_i; or (3) update the priority encoder.

Let us first consider a prefix match subfield. The first step for rule modification is to construct the up-to-date bit vectors 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 [12]. Notice 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 resides. In each PE, the logic-based rule decoder performs Algorithm 1 to update the memory content by itself.

Algorithm 1 Up-to-date bit vectors for subfield j

Input n ternary strings each of s bits: $T_{i,j}$, where $T_{i,j} \in \{0, 1, *\}$, $i = 0, 1, ..., n - 1$.
Output 2^s bit vectors each of length n:
$b_j^{(k_j)} = B_{j,0}^{(k_j)} B_{j,1}^{(k_j)} ... B_{j,n-1}^{(k_j)}$, where $B_{j,i}^{(k_j)} \in \{0, 1\}$,
$k_j = 0, 1, ..., 2^s - 1$, and $i = 0, 1, ..., n - 1$.

1: **for** $i = 0, 1, ..., n - 1$ **do**
2:    **for** $k_j = 0, 1, ..., 2^s - 1$ **do**
3:       **if** k_j matches the regular expression $T_{i,j}$ **then**
4:          $B_{j,i}^{(k_j)} \leftarrow 1$
5:       **else**
6:          $B_{j,i}^{(k_j)} \leftarrow 0$
7:       **end if**
8:    **end for**
9: **end for**

In our approach, we replace all the bit vectors in a BV array with the up-to-date bit vectors. An alternative can be to update bit vectors in a BV array selectively only if the up-to-date and outdated bit vectors are different; however, this technique requires extra comparison units, which complicates our design of each PE and consumes more resources.

Notice k_j is directly used as the input address to the data memory storing a BV array, and the bit vectors are stored row-by-row in the data memory. To modify a single rule, we need 2^s memory write accesses, each access modifying a single bit vector. Multiple rules can be modified concurrently, but in the worst case, only a single rule is modified during one update of the BV array.

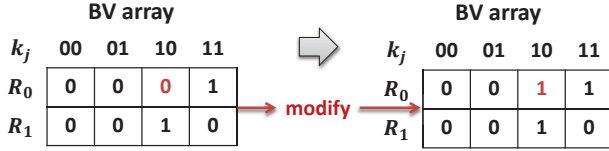We show an example for rule modification in a particular

BV array

k_j	00	01	10	11
R_0	0	0	0	1
R_1	0	0	1	0

→ **modify** →

BV array

k_j	00	01	10	11
R_0	0	0	1	1
R_1	0	0	1	0

Figure 7: Modifying a rule R_0 ($n = 2$, $s = 2$)

BV array

k_j	00	01	10	11
R_0	0	0	0	1
R_1	0	0	1	0

	Valid bit
R_0	0
R_1	1

→ **update** →

BV array

k_j	00	01	10	11
R_0	0	0	0	1
R_1	0	0	1	0

	Valid bit
R_0	0
R_1	0

Figure 8: Deleting an old rule R_1 ($n = 2$, $s = 2$)

BV array

k_j	00	01	10	11
R_0	0	0	0	1
R_1	0	0	1	0

→ **modify** →

BV array

k_j	00	01	10	11
R_0	0	0	0	1
R_1	1	1	0	0

	Valid bit
R_0	1
R_1	0

→ **update** →

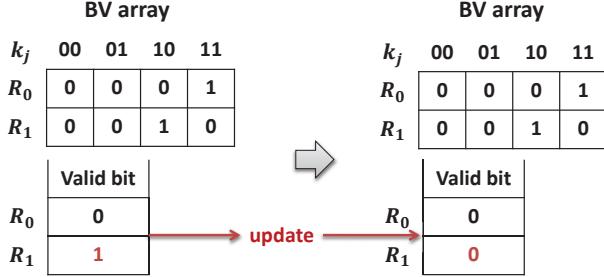|	Valid bit
R_0	1
R_1	1

Figure 9: Inserting a new rule R ($n = 2$, $s = 2$)

subfield j in Figure 7. In this example, R_1 is to be updated; the ternary string representation of R_1 needs to be changed from "11" to "1*". We have an outdated BV array to be updated due to this rule modification. We replace the entire BV array, although there is only 1-bit difference (for $k_j = 10$ and R_0) between the outdated BV array and the up-to-date BV array. We repeat this operation to other subfields requiring prefix match in order to update all the outdated BV arrays.

The update process for a range match field does not require any bit vectors to be constructed, therefore the range rules can be updated by modifying the memory contents directly. If the update process requires to change the priority of the old rule, i.e., the new rule and the old rule have different priority orders, we update the priority encoders based on a dynamic tree structure [20]; the time complexity to update the dynamic tree is $O(\log N)$. In general, if a prioritized rule set requires both prefix match and range match to be performed, the parallel time complexity for modifying a rule is $O(\max[2^s, 2, \log N])$.

5.3 Deletion

After the RID check, rule deletion can be performed as: Given a RID R ($\exists i \in \{0, 1, ..., N-1\}$ such that $R_i = R$, i.e., RID R already exists in the rule set), delete the rule with RID R_i from the rule set. i.e., R_i should no longer be used to 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 Figure 8. In this example, initially R_0 is invalid; R_1 is valid and to be deleted. During the update, 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
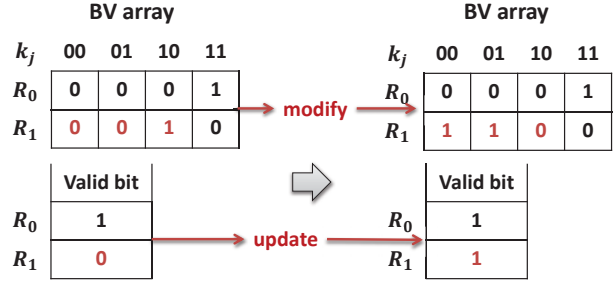
the horizontal pipeline. As a result, if a rule is invalid (i.e. its valid bit is "0"), the corresponding position for this rule in the final AND result can only be "0", indicating the input does no match this rule.

5.4 Insertion

After the RID check, rule insertion can be performed as: Given a rule with RID R ($R_i \neq R$, for $\forall i \in \{0, 1, ..., N-1\}$, i.e., RID R does not exist in the rule set), 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 with RID R.

To insert a rule, we first check whether there is any invalid rule in the rule set; we denote this process as *validity check*. 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 algorithms presented in Section 5.2. Finally, we validate this new rule by updating its corresponding valid bit.

Figure 9 shows an example of rule insertion in a subfield requiring prefix match. In this figure, rule R_1 is invalid as indicated by the valid bit; all of the bit vectors in the BV array are to be replaced by the up-to-date bit vectors. We validate this new rule R by setting the valid bit of R_1 to "1".

5.5 Architecture for Dynamic Updates

5.5.1 Storing valid bits

The data memory in the modular PE with striding and clustering techniques in Figure 6 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 2-dimensional pipelined architecture. The reason of doing this is: 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 $\lceil \frac{N}{s} \rceil$ rows and $(\lceil \frac{L}{s} \rceil + 1)$ columns, where valid

Table 3: Update overhead (clock cycles) in a PE[l, 0], $l = 0, 1, ..., \lceil \frac{N}{n} \rceil - 1$

Update types	Modify	Delete	Insert
RID check	1	1	1
Rule translation	1	1	1
Validity check	-	0	0
Updating BV array/ range boundaries	$2^s/$ 2	-	$2^s/$ 2
Updating priority	$\log N$	$\log N$	$\log N$
Updating valid bits	-	1	1
Worst-case total overhead	2+ $\max[2^s, 2, \log N]$	3+ $\log N$	3+ $\max[2^s, 2, \log N]$

bits are stored and extracted in PE[l, 0], $l = 0, 1, ..., \lceil \frac{N}{n} \rceil - 1$ (the first column).

5.5.2   Logic-based rule decoder

To save I/O pins on FPGA, we use the pins for packet header (in total L pins) to input a new rule. In each PE[l, 0], $l = 0, 1, ..., \lceil \frac{N}{n} \rceil - 1$, the RID of the rule that needs an update is provided to the rule decoder. During an update, the control signals are generated by the rule decoder. For PE[l, 0], $l = 0, 1, ..., \lceil \frac{N}{n} \rceil - 1$, the up-to-date valid bits are also generated by the rule decoders. Specifically, the rule decoder is in charge of:

1. RID check (for all update operations, only in PEs of the first column): The rule decoders check whether the RID of the new rule already exists in the rule set.

2. Rule translation (for modification/insertion, in all PEs): Based on the new rule, all the data to be written to the data memory are generated by the rule decoder; the data to be written can be 2^s n-bit vectors in a BV array for a prefix rule, or 2 s-bit range boundaries for a range rule.

3. Validity check (for deletion/insertion, only in PEs of the first column): The validity check is also performed by the rule decoder. In our implementation, the rule decoder reports the position of the first invalid bit in the data memory.

4. Construction of up-to-date valid bits (for insertion, only in PEs of the first column): The rule decoder also provides the control signals and the data to be written back (up-to-date valid vector) to the data memory.

The rule decoder is a distributed controller for each PE; it gives each PE the ability to reconfigure its memory contents by itself during an update (self-reconfiguration). As a result, the dynamic updates are performed in a fine-grained distributed manner in parallel.

5.6   Update Schedule and Overhead

To further improve the performance of our architecture, we overlap the validity check process with on-going packet header match process; the validity check results are reported every clock cycle to the rule decoder for all PE[l, 0], $l = 0, 1, ..., \lceil \frac{N}{n} \rceil - 1$. Table 3 summarizes the overheads for update operations in such a PE:

- For all update operations, the RID of the new rule has to be checked against all rules, which results in 1-cycle overhead (pipeline bubble).
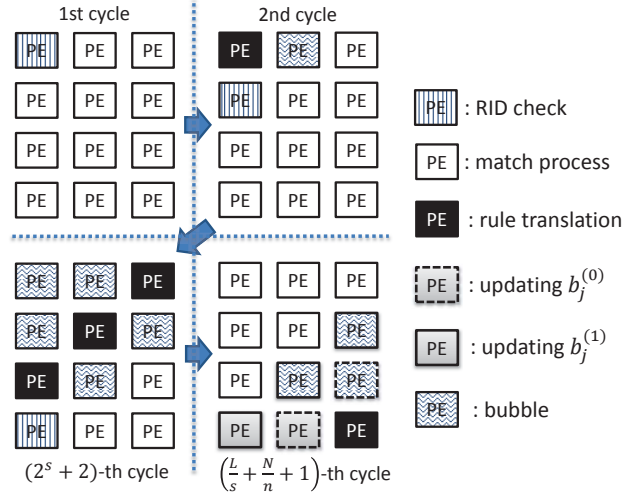

Figure 10: Example: modifying a rule ($\frac{L}{s} + 1 = 3$, $\frac{N}{n} = 4$, $s = n = 1$)

- The rule decoder generates control signals based on the input rule provided on the s packet header input wires; hence the rule translation cannot be overlapped with the packet header match process. In our implementation, this leads to a 1-cycle overhead.

- To update (1) the BV array or (2) the range boundaries, the rule decoder either (1) initiates 2^s memory write accesses or (2) initiates 2 memory write access. During the update, the memory cannot be read by the packet header match process. In addition, 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.

- The update process for the valid bits (n memory write access) cannot be overlapped with the on-going packet header match process; this results in another 1-cycle overhead.

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 Figure 10. In this example, the RID check results indicate the RID of the new rule exists in the last horizontal pipeline, therefore only the PEs in the last row require the contents in the data memories to be updated. As can be seen, although ($\frac{L}{s} + \frac{N}{n}$) clock cycles are required for the new rule to propagate across the entire 2-dimensional pipelined architecture, this amount of time does not contribute to the total update overhead of the entire architecture; this is because the update is performed in a distributed manner. Assuming $2^s \geq \log N \geq 2$, the total number of bubbles injected into a particular PE is at most 2^s for a rule modification. For example, in Figure 10, $2^s = 2$ bubbles are injected into each of the horizontal pipelines except the last one, while the packet header match process is stalled for a total number of $2^s + 2 = 4$ clock cycles for each PE.

Since all other PEs except PE[l, 0], $l = 0, 1, ..., \lceil \frac{N}{n} \rceil - 1$ neither perform validity check nor update valid vectors, the

PEs in the first column of the architecture introduce the most update overhead. Hence Table 3 also lists the worst-case update overhead for all PEs. As can be seen, the insert operation introduces the most overhead among all types of update operations.

6. PERFORMANCE EVALUATION

6.1 Experimental Setup

We conducted experiments using Xilinx ISE Design Suite 14.5, targeting the Virtex 6 XC6VLX760 FFG1760-2 FPGA [19]. This device has $118,560$ logic slices, 1200 I/O pins, 26 Mb BRAM (720 RAMB36 blocks), and can be configured to realize large amount of distRAM (up to 8 Mb). All the memory modules are configured as dual-ported for read access to enhance the throughput. It has 2 slices in a Configurable Logic Block (CLB), each slice having 4 LUTs and 8 flip-flops. Clock rate and resource consumption are reported using post-place-and-route results.

Since the construction of bit vectors or range boundaries does not explore any rule set features[6], the performance of the architecture does not depend on rule set features other than the rule set size. We use randomly generated bit vectors; we also generate random packet headers for both 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 input bits (L). The number of rules in a rule set is chosen to range from 128 to 1K, since most of the real-life rule sets are fairly small [9].

6.2 Design Parameters/Performance Metrics

We vary several design parameters to optimize our architecture, including:

Size of the rule set (N) The total number of rules in a rule set

Length of the bit vector (n) The number of bits in a bit vector

Length of the input (L) The total number of input packet header bits

Stride (s) The number of bits for a subfield in the packet header

Update rate (U) The total number of all update operations (modify, delete or insert) 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 input packet bits (assuming the minimum packet size to be $\max\{L, 40\,bytes\}$) processed per second without any update operation

Sustained throughput ($T_{sustained}$) The actual number of packet bits (including both header and payload) processed per second considering all update operations

---

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

Table 4: Clock rate (MHz) of various designs

		Stride (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

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 for classifying an incoming packet [21]

6.3 Empirical Optimization of Parameters

We first optimize our design for given N and L, and varying n and s. We choose a small value of $N = 128$ at first, and $L = 356$ for OpenFlow packet classification. The values of n and s which give the best performance are then used in later designs with other values of N and L; we scale our architecture with respect to N, and also show the performance for classic packet classification.

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 ($200 \sim 400$ MHz) with small variations in various designs. They correspond to high throughput for OpenFlow packet classification ($128 \sim 256$ Gbps).

- For $s = 1, 2$, BV arrays are stored in 2^s-input "shallow" memories. This memory organization underutilizes the distRAM modules on FPGA (6-input LUT based). In addition, since we have a large number of pipeline stages for $s = 1, 2$, the entire architecture consumes a large amount of registers; the complex routing between these registers also limits the maximum achievable clock rate ($200 \sim 300$ MHz).

- For $s = 3, 4, 5, 6$, the best performance is achieved for either $n = 8$ or $n = 16$. Notice 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 8 flip-flops of a slice to register a bit vector, while a PE with $n = 16$ uses exactly all 16 flip-flops in a CLB to register a bit vector; these two configurations introduce the least routing overhead.

- 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 the deterioration of the clock rate.

- We achieve the best performance for our architecture when $s = 4$ and $n = 8$. This is because all the LUTs within a single slice can be configured as 128-bit dual-port distRAM; the configuration of $n = 8$ and $s = 4$ not only uses up all the 8 flip-flops in a slice, but also
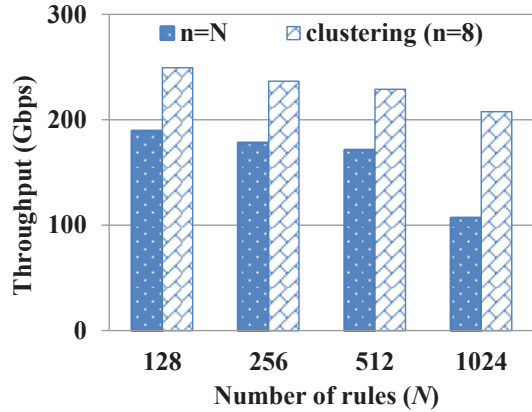
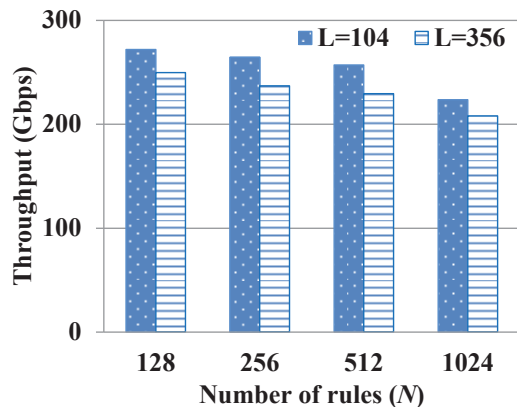Figure 11: Scalability with respect to N

Figure 12: Scalability with respect to L

Figure 13: Sustained throughput

Table 5: Resource consumption ($s = 4$, $n = 8$ and $L = 356$)

No. of rules N	128	256	512	1024
No. of logic slices	14773	29056	57209	112812
(% of total)	(12%)	(25%)	(48%)	(95%)
No. of I/O pins	722	723	724	725
(% of total)	(60%)	(60%)	(60%)	(60%)

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 the architecture for other values of N and L.

6.4 Scalability of Throughput

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

Figure 11 shows the throughput of our architecture with respect to various values of N. As can be seen, our architecture maintains very high clock rate (324 MHz) and throughput (208 Gbps) even for $N = 1024$. We also show in the same figure the necessity of using the clustering technique discussed in Section 4.4. Compared to the case with no clustering technique (the basic pipelined architecture with $n = N$), our architecture achieves much better throughput performance (up to $2\times$) when the rule set is large; in our architecture, the clock rate drops much slower as N increases.

Figure 12 shows the achieved throughput for both the classic packet classification ($L = 104$) and OpenFlow packet classification ($L = 356$). We integrated range match to the port number fields for the classic packet classification. Our architecture achieves high throughput for the classic packet
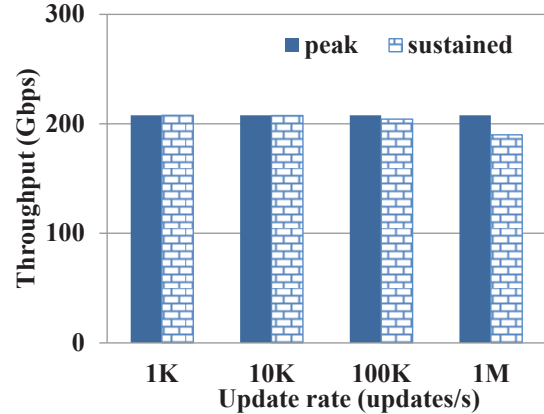
classification; the performance is also achieved with little degradation for OpenFlow packet classification.

6.5 Updates and Sustained Throughput

As discussed in Section 5.6, 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:

$$T_{sustain} = T_{peak} \cdot \frac{f - 2 \cdot (\max[2^s, 2, \log N] + 3) \cdot U}{f} \quad (1)$$

where f denotes the maximum clock rate achieved for a specific design. The factor of 2 comes from the fact that memory write access on FPGA can only be configured as single-ported instead of dual-ported.

Figure 13 shows the sustained throughput for our architecture, considering the worst-case scenario for all update operations. In the implementation, $s = 4$ and $n = 8$ are used to achieve the best clock rate for $N = 1024$. As can be seen, our architecture sustains a high throughput of 190 Gbps with 1M updates/s, although 1M updates/s is pessimistic considering real-world traffic.

6.6 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 the number of rule 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. Figure 14 shows a comparison of our approach with existing solutions on FPGA. We consider a 15-field OpenFlow classification rule
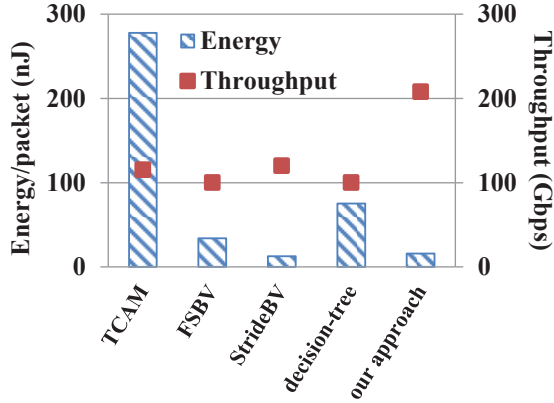
Figure 14: Energy efficiency

set with 1K rules for all the schemes. Since the known implementations of TCAM, FSBV and StrideBV approaches do not support range match, we assume all the 1K rules only require prefix match to be performed. 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 [22]; we ignore the power consumed by the extra logic for managing the TCAM access. We scale up the memory consumption of FSBV and StrideBV to estimate the total power consumed for OpenFlow packet classification; the power consumption for FSBV and StrideBV, as well as our approach, is evaluated using XPower Analyzer tool available in the Xilinx ISE Design Suite 14.5 on the state-of-the-art Virtex 6 XC6VLX760 FFG1760-2 FPGA. The decision-tree based approach on FPGA [23] is also scaled to the same Virtex 6 device. For StrideBV, the most energy-efficient design with $s = 4$ [9] is considered, while the energy consumption of our approach is based on the design with $s = 4$ and $n = 8$.

We notice that our design runs at a higher clock rate than other FPGA-based designs including the FSBV, StrideBV and decision-tree based approaches; to make a fair comparison, we also show the throughput for all the approaches in Figure 14. 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 have the following observations:

- All the FPGA-based approaches for OpenFlow packet classification achieve at least 4× energy efficiency than the TCAM solution.

- Compared with 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 [24] on FPGA.

- Compared with 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 with StrideBV (12.7 nJ/packet), our approach consumes more energy (1.25×). This is because 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× compared with other approaches) and supports fast incremental update.

6.7 Summary

We summarize the advantages of the proposed architecture as follows:

1. **Parameterized**: The architecture is highly parameterized; it can be optimized with respect to various performance metrics.

2. **Efficient support for range match**: The architecture supports efficient range match without any range-to-prefix conversion.

3. **Rule-set-independent**: The performance does not depend on any rule set features other than the rule set size.

4. **High-throughput**: All the PEs access their designated distRAM modules independently. The memory access is localized, resulting in shorter wires connecting the AND gate and the memory modules in each PE. This leads to high clock rate and high throughput on FPGA.

5. **Scalable with respect to rule set size**: The longest wire length is not significantly affected by the increasing total number of rules; the architecture sustains high throughput for a large number of rules, assuming we have sufficient hardware resources.

6. **Scalable with respect to input length**: The throughput performance is not adversely affected by the increasing number of the input packet header bits. Our architecture achieves good performance for both classic and OpenFlow packet classification.

7. **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.

8. **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 per packet.

7. CONCLUSION AND FUTURE WORK

In this paper we presented a 2-dimensional pipelined architecture for both the classic and OpenFlow packet classification. The resulting architecture employs localized memory access on FPGA and achieves scalability with respect to

both the rule set size and the input length. This architecture also supports efficient range search and does not require any knowledge of the rule set features; it supports very high throughput with little dynamic update overhead. We also compared the energy efficiency of this architecture with existing solutions and demonstrated high energy efficiency.

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.

8. REFERENCES

[1] 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.

[2] "OpenFlow Switch Specification V1.1.0," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf.

[3] "OpenFlow Switch Specification V1.3.1," https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.

[4] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.

[5] 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, 2005.

[6] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," in *Proc. ACM SIGCOMM*, 2005, pp. 193–204.

[7] 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. of the 2007 ACM Symposium on Applied Computing (SAC)*, 2007, pp. 264–269.

[8] 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, 2012.

[9] T. Ganegedara and V. K. Prasanna, "StrideBV: Single Chip 400G+ Packet Classification," in *13th IEEE International Conference on High Performance Switching and Routing (HPSR)*, 2012, pp. 1–6.

[10] P. Gupta and N. McKeown, "Dynamic Algorithms with Worst-Case Performance for Packet Classification," in *Proc. of the IFIP-TC6*, 2000, pp. 528–539.

[11] B. Vamanan and T. N. Vijaykumar, "TreeCAM: Decoupling Updates and Lookups in Packet Classification," in *Proc. of the 7th COnference on emerging Networking EXperiments and Technologies(CoNEXT)*, 2011, pp. 27:1–27:12.

[12] W. Jiang and V. K. Prasanna, "Field-split Parallel Architecture for High Performance Multi-match Packet Classification using FPGAs," in *Proc. of the 21st Annual Symp. on Parallelism in Algorithms and Arch. (SPAA)*, 2009, pp. 188–196.

[13] D. E. Taylor, "Survey and Taxonomy of Packet Classification Techniques," *ACM Computing Surveys*, vol. 37, no. 3, pp. 238–275, 2005.

[14] W. Jiang and V. K. Prasanna, "Large-scale Wire-speed Packet Classification on FPGAs," in *Proc. of the ACM/SIGDA intl. symposium on Field Programmable Gate Arrays (FPGA)*, 2009, pp. 219–228.

[15] D. E. Taylor and J. S. Turner, "Scalable Packet Classification using Distributed Crossproducting of Field Labels," in *Proc. IEEE INFOCOM*, 2005, pp. 269–280.

[16] V. Pus, J. Korenek, and J. Korenek, "Fast and Scalable Packet Classification using Perfect Hash Functions," in *Proc. of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2009, pp. 229–236.

[17] T. V. Lakshman and D. Stiliadis, "High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching," in *Proc. ACM SIGCOMM*, 1998, pp. 203–214.

[18] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and Scalable Layer Four Switching," in *Proc. ACM SIGCOMM*, 1998, pp. 191–202.

[19] "Virtex-6 FPGA Family," http://www.xilinx.com/products/virtex6/index.htm.

[20] Y.-H. E. Yang and V. K. Prasanna, "High Throughput and Large Capacity Pipelined Dynamic Search Tree on FPGA," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2010, pp. 83–92.

[21] A. Kennedy, X. Wang, and B. Liu, "Energy Efficient Packet Classification Hardware Accelerator," in *Proc. of IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008, pp. 1–8.

[22] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-efficient TCAMs for Forwarding Engines," in *Proc. IEEE INFOCOM*, vol. 1, 2003, pp. 42 – 52.

[23] W. Jiang and V. K. Prasanna, "Scalable Packet Classification on FPGA," *IEEE Trans. VLSI Syst.*, vol. 20, no. 9, pp. 1668–1680, 2012.

[24] R. Balasubramonian, N. Muralimanohar, K. Ramani, L. Cheng, and J. Carter, "Leveraging Wire Properties at the Microarchitecture Level," *Micro, IEEE*, vol. 26, no. 6, pp. 40–52, 2006.