Finding Nonequivalent Classifiers in Boolean Space to Reduce TCAM Usage

Rihua Wei, *Member, IEEE*, Yang Xu, *Member, IEEE*, and H. Jonathan Chao, *Fellow, IEEE*

***Abstract*—Packet classification is one of the major challenges today in designing high-speed routers and firewalls, as it involves sophisticated multi-dimensional searching. Ternary Content Addressable Memory (TCAM) has been widely used to implement packet classification, thanks to its parallel search capability and constant processing speed. However, TCAMs have limitations of high cost and high power consumption, which ignite the desire to reduce TCAM usage. Recently, many works have been presented on this subject due to two opportunities. One is the well-known range expansion problem for packet classifiers to be stored in TCAM entries. The other is that there often exists redundancy among rules. In this paper, we propose a novel technique called *Block Permutation (BP)* to compress the packet classification rules stored in TCAMs. Unlike previous schemes that compress classifiers by converting the original classifiers to semantically equivalent classifiers, the BP technique innovatively finds semantically nonequivalent classifiers to achieve compression by performing block-based permutations on the rules represented in Boolean Space. We have developed an efficient heuristic approach to find permutations for compression and have designed its hardware implementation by using Field-Programmable Gate Array (FPGA) to preprocess incoming packets. Our experiments with ClassBench classifiers and Internet Service Provider (ISP) real-life classifiers show that the proposed BP technique can significantly reduce 31.88% TCAM entries on average, in addition to the reduction contributed by other state-of-the-art schemes.**

Index Terms*—Classifier Minimization, Logic Optimization, Packet Classification, Ternary Content-Addressable Memory (TCAM), Field-Programmable Gate Array (FPGA).

I. Introduction

Packet classification is used as a basic building block in many network applications, such as quality of service (QoS), flow-based routing, firewalls, and network address translation (NAT) [1][2]. In packet classification, information is extracted from the packet header and compared against a classifier consisting of a list of rules. Once an incoming packet matches some rules, it will be processed based on the action associated with the highest-priority matched rule.

Manuscript received December 20, 2012; revised September 16, 2013 and May 02, 2014; accepted January 19, 2015; approved by IEEE/ACM Transactions on Networking Editor I. Keslassy. A preliminary version of this paper [26] was presented in part at the IEEE INFOCOM, mini-conference, Orlando, Florida, USA, March 2012.

R. Wei is currently with Tensorcom Inc., USA. This work was conducted while he was with New York University Polytechnic School of Engineering. (e-mail: rchwei@gmail.com). Y. Xu and H. J. Chao are with New York University Polytechnic School of Engineering, Brooklyn, NY 11201 USA. (e-mail: {yang, chao}@nyu.edu).

Table I gives a sample classifier with three rules, in which each rule specifies a pattern of five fields: source IP and destination IP (prefixes), source port and destination port (ranges), and protocol type. According to the traditional viewpoint, a packet classifier consists of five dimensions due to the five fields. But in this paper, from the geometric point of view, we treat each rule as a block in the 104-dimensional Boolean Space corresponding to the 104 bits in the five fields.

TABLE I
A Sample Packet Classifier

Rule	Source IP	Dest IP	Source Port	Dest Port	Protocol	Action
R1	*	192.168.1.1	[1, 5]	[1, 5]	UDP	Accept
R2	166.111.*.*	192.168.1.*	*	*	*	Deny
R3	*	*	*	*	*	Deny

Ternary Content Addressable Memory (TCAM) is widely used to implement packet classification because of its parallel search capability and constant processing speed. A TCAM has a massive array of entries [3], in which each bit can be represented as ‘0’, ‘1’, or ‘*’ (wildcard). Before a rule can be stored in TCAMs, its range fields have to be converted to prefixes. This would cause the well-known *range expansion* problem. For example, rule *R2* in Table I requires only one TCAM entry since it doesn’t contain any range in all fields. But for rule *R1*, both the source port and destination port contain a range [1, 5]. So both of them need to be expanded to three prefixes, i.e., “001”, “01*”, and “10*”. The combination of the prefix specifications of the two ranges makes *R1* consumes $3 \times 3 = 9$ TCAM entries. Besides, there often exists redundancy among rules. For example, *R2* is actually unnecessary and can be safely removed from the classifier, because it is completely covered by *R3*. These two problems lead to inefficiency in TCAM use. Because TCAMs are expensive and power-hungry, it is very important to reduce the number of TCAM entries required to represent a classifier.

Previous works on TCAM reduction can be classified into two categories, *Range-Field Optimization* and *All-Field Optimization*. Range-field optimization schemes focus only on the source port field and destination port field to address the range expansion problem. Normally, the best compression of a range-field optimization scheme is to reduce an expanded classifier to the size before expansion. In this category, [4] attempts to modify TCAM hardware architecture to support range matching; [6][7] replace each range with a new binary code to avoid range expansion; [5][9][25] try to expand a range to a minimum number of binary codes; [8] first transforms ranges to simplified ones, and then finds the minimum expansion on the new ranges. The novelty of [8] is that it can optimize rule length (some related works can also be found in [10][11]). In the category of all-field optimization, proposed schemes [12][13] [14][24] work on all the five fields to address both the aforementioned problems based on the optimization of ternary strings, which can be prefix or arbitrary ternary string. The common part of all these schemes is to first expand all ranges to prefixes, getting a new classifier with no range fields, and then convert the non-range classifier to a semantically equivalent one that consumes fewer TCAM entries. Because all-field optimization schemes consider both the range expansion and the redundancy among rules, they are able to find a classifier that is smaller than the original one before range expansion.

The new technique proposed in this paper, *Block Permutation (BP)*, is also an all-field optimization solution. Distinct from previous schemes, BP finds semantically nonequivalent classifiers. More specifically, it maps rules to blocks in Boolean Space and swaps blocks (so called block permutation, please refer Section V for a formal definition) to seek opportunities of merging blocks. In BP, the incoming packets need to be preprocessed before being compared against the compressed classifier in TCAM. This preprocessing can be implemented by FPGA. In this paper, we propose an efficient heuristic algorithm to find permutations and present an FPGA implementation methodology. Our experiments on ClassBench [15] classifiers and ISP classifiers show that the BP technique can significantly reduce TCAM entries.

The rest of this paper is organized as follows: Section II reviews related works in detail and gives the motivation for BP. Section III introduces BP technique with a warm-up example. Section IV formally defines the BP problem and analyzes its complexity. Section V proposes the heuristic BP algorithm to compress classifiers. Section VI proposes an FPGA implementation methodology. Section VII discusses the strategy of classifier update. Section VIII presents the experimental results. Finally, Section IX concludes the paper.

II. Related Work and Motivation

Previous schemes find semantically equivalent classifiers to reduce TCAM usage by taking advantage of two properties:

1) *Action-Oriented.* In packet classification, we are interested in the action rather than the ID associated with the matched rule. Therefore, we can modify a classifier as long as the modification doesn't change the action returned by each classification operation. Consider Fig. 1(a) as an example. We can merge *R1* and *R2* into *R4*. Though the rules have been changed, the compressed classifier is still equivalent to the original classifier; i.e. the compressed classifier can still report the same action as the original classifier does. This property actually embodies the principle of Boolean logic optimization.

2) *First-Matching.* When multiple rules match the same packet, TCAM reports only the first matched rule. Fig. 1(b) and (c) are two examples showing how to compress a classifier using the first-matching property. In Fig. 1(b), because *R2* is completely covered by *R1*, any packet matching *R2* will definitely match *R1* as well. So, *R2* is a redundant rule. Removing it will not affect the packet classification results. In Fig. 1(c), *R1*, *R2*, and *R3* cannot be directly merged. By adding *R0*, we can merge all of them with *R0* into a single rule *R6*. But this would result in a non-equivalent classifier, because *R0* forces the action of "0000" to be "Accept", while it is "Deny" in the original classifier. To make the classifiers equivalent, we add *R5* above *R0*. Because of the first-matching property, *R0* will be blocked by *R5*, keeping the classifiers equivalent.

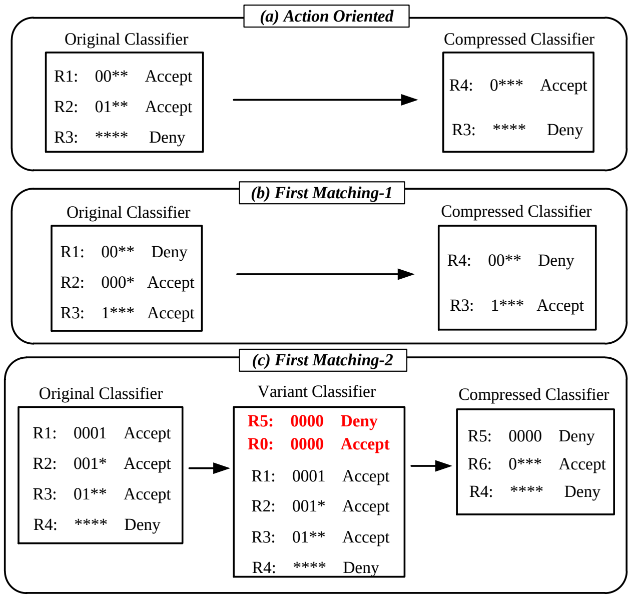


Fig. 1. Scenarios of the two properties for prefix compression. (a) Action Oriented. (b) First Matching -1. (c) First Matching -2.

These two properties have been used in both range-field optimization and all-field optimization. In the category of range-field optimization, [5] first expands a range to *binary-reflected gray codes* and then optimizes the codes. The second step of [5] is actually a case of logic optimization. [9][25] propose optimal solutions of range expansion and also embody the idea of logic optimization and the first-matching property. In the category of all-field optimization, Dong *et al.* in [12] proposed four simple heuristic algorithms to find equivalent classifiers consuming fewer TCAM entries. Dong's algorithms are also special cases of logic optimization and the first-matching property. Liu *et al.* in [13] propose an algorithm called *TCAM Razor*, which uses *Firewall Decision Diagram (FDD)* to convert a multi-field rule to multiple one-field rules in a hierarchy. Because TCAM Razor can guarantee that a range in one field expands to the optimal number of prefixes based on the first-matching property, it achieves a better compression than [12] does.

While [5][9][25] considers only range fields, [12][13] consider all fields in a classifier. However, [12][13] work on each field individually and do not explore the compression across different fields. *Bit Weaving* in [24] is the first all-field optimization scheme attempting to break the boundaries of fields. It can find and merge two rules with one bit different, no matter in which field the bit is located. Another significant cross-field solution is proposed in [14] by McGeer *et al.* In that solution, the classifier compression problem is a special logic optimization problem with 104 variables, where each rule in the classifier is a product of several variables in Boolean representation (or say, a block in Boolean Space). Therefore, the existing logic optimization techniques can be applied to compress classifiers by using the action-oriented property. Moreover, with the first-matching property of TCAM, the compression can be even better [14].

Fig. 2 is an example of McGeer's scheme [14], which consists of three steps. In the first step, the original classifier (with six rules) is mapped into the Boolean Space shown in the *Karnaugh Table* [16]. Each rule corresponds to a block (or a point) in the table. During the mapping, the overlapping portion of rules is associated with the action of the highest-priority rule. For example, the point in the upper-left corner (i.e., "0000(WXYZ)"), which is covered by both the first and the last rules, is assigned with the action of the first rule. In the second step, classical logic optimization algorithms are applied in the Karnaugh Table to merge the neighboring points with the same action to reduce the number of rules (i.e. to merge "Accept" points and get *Classifier 1* in the example). In the third step, the first-matching property is applied to *Classifier 1*. This application is similar to the example in Fig. 1 (c), and as a result, we can get an even smaller classifier that has only three rules (*Classifier 2*). Please note that all three classifiers in the example are semantically equivalent since they correspond to the same Karnaugh Table.

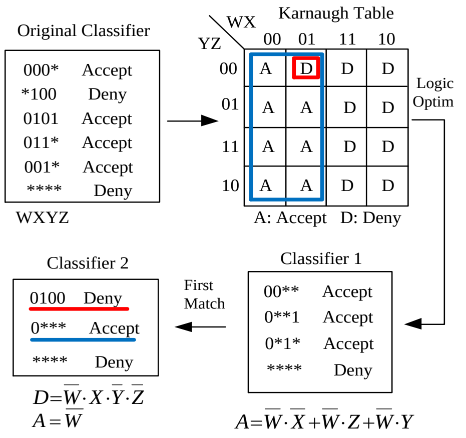


Fig. 2. An example of the scheme proposed by McGeer [14]

McGeer's scheme [14] is indeed representative of previous schemes. First, it is a pure bit-level solution that breaks the boundaries of fields and exploits compression opportunities in every bit of a classifier. Logic optimization at the bit-level is, by nature, better than that limited to a field. Second, it takes advantage of the first-matching property. However, the performance of this scheme greatly depends on the *rule distribution* of a classifier (i.e., the distributions of rules with action "Accept" or "Deny" in the Boolean Space).

In Fig. 3, we show two classifiers in different rule distributions. In Fig. 3 (a), *rule elements* associated with the same action are densely populated (here, a *rule element* is the smallest unit, i.e., a point in the Boolean Space associated with an action), while in Fig. 3 (b), rule elements are spread sparsely in the Boolean Space. Obviously, logic optimization and the first-matching property are very suitable for handling the densely populated rule distribution in Fig. 3 (a). In contrast, logic optimization and the first-matching property perform badly under the rule distribution in Fig. 3 (b). This is because rule elements in Fig. 3 (b) are spread sparsely and no any two neighboring rule elements have the same action; thus, no two elements can be directly merged using logic optimization. Therefore, under such circumstances, logic optimization cannot contribute much compression. Neither can the first-matching property. Because, for example, to reduce the number of "Accept" rules using the first-matching property, we have to create and put many "Deny" rules in the high-priority places (similar to what we have done in Fig. 1 (c)). This case would result in an even larger classifier.

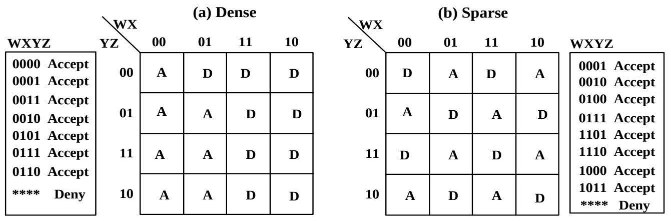


Fig. 3. Typical Rule Distributions (a) Dense (b) Sparse

The above observation motivates us to develop the BP technique to compress classifiers in sparse rule distributions like the one shown in Fig. 3 (b). BP first converts sparse rule distributions to dense rule distributions by swapping blocks (or points), and then applies the logic optimization to merge rule elements that cannot be merged originally. This technique is a good complement for previous schemes.

Similar to McGeer's scheme [14], the BP technique is also a bit-level solution, except that BP swaps blocks (or points) to generate a nonequivalent classifier and thus needs preprocessing on incoming packets. Bit Weaving [24] also swaps bits of rules in a classifier. But its purpose is not to change rule distribution and the finial output is still an equivalent classifier, which is fundamentally different from BP. Layered Interval Coding in [6] also needs preprocessing on incoming packets. But it is not based on logic optimization and the first-matching property, and requires extra bits in each TCAM entry. Its working conditions are different from BP's.

In the rest of the paper, we will present the details of the BP technique. For convenience, all the classifiers in the examples consist of several "Accept" rules followed by a "Deny" rule as the default rule. For simplicity, all rules consist of only 4 bits, which are denoted by W, X, Y, and Z, respectively. We always assume that the default order of bits is WXYZ. So, denotation like Point "0000(WXYZ)" will be simplified to "0000".

III. A WARM-UP EXAMPLE OF BP

We use a simple example in Fig. 4 to demonstrate the main idea of BP. In the example, the *Original Classifier* will be compressed by applying two simple permutations to convert the classifier's rule distribution from sparse to dense.

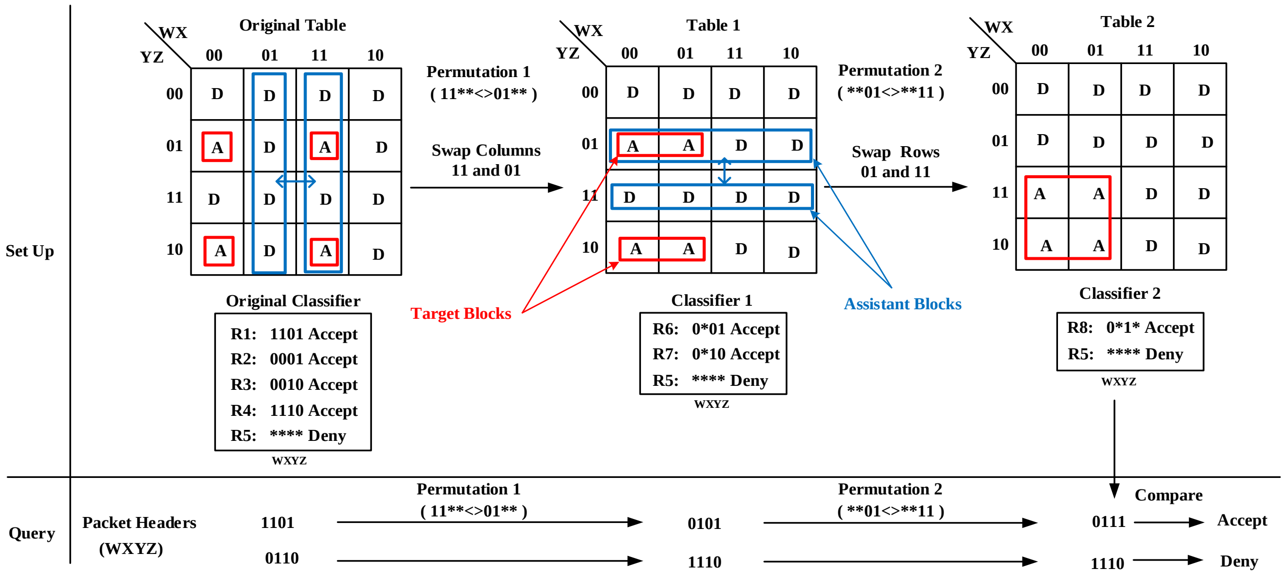


Fig. 4. An example of the BP technique

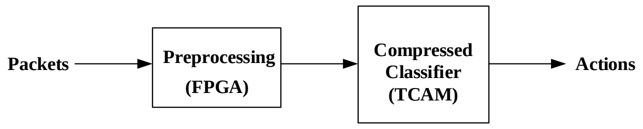


Fig. 5. The architecture of BP implementation

The two permutations in Fig. 4 are denoted as “11**<>01**” and “**01<>**11” in order. The first permutation “11**<>01**” is to swap Columns “11” and “01” in the *Original Table*. In this permutation, counterpart points in the two columns exchange their positions. For example, “0100” exchanges with “0111”. As a result, we get *Table 1*. In the second permutation “**01<>**11”, we swap Rows “01” and “11” in *Table 1* and get *Table 2*. Then by applying logic optimization in *Table 2*, the original five rules are merged into two rules. Finally, we get a compressed classifier that will be stored in a TCAM as Fig. 5 shows.

When a packet comes for query, correspondingly, we need to apply the same permutations to the header of the packet, which is the preprocessing step as shown in Fig. 5. In the first permutation, if the WX bits of the packet header are “11” (or “01”), we change them to “01” (or “11”); otherwise, we keep the WX bits unchanged. In the second permutation, similar processing is done on YZ bits. Obviously, by using the preprocessed packet headers to look up *Classifier 2*, we get the same actions as those we get when using the original packet headers to search the *Original Classifier*.

For the practicality of this BP scheme, we need to consider the following issues:

1) *Compression Performance*. In L-dimensional Boolean Space, the best compression that BP can do is to move all “Accept” points (or blocks) together and merge them into a block, which can be represented by a range. According to [9][25], that range expands to at least L entries. If there are more “Accept” points than “Deny” points, we can turn to move “Deny” points, then the number can be as low as $\lceil (L+1)/2 \rceil$.

2) *Overhead*. While BP can reduce the TCAM size, the preprocessing does introduce overhead. But the overhead can be much smaller than that the TCAM resource saved.

Let’s use *Permutation 2* in Fig. 4 (i.e., “**01<>**11”) as an example that can save one TCAM entry, to explain how to compute the overhead. Suppose that before the permutation, the packet header has four bits, which are $W^{(1)}$, $X^{(1)}$, $Y^{(1)}$, and $Z^{(1)}$. After that, the four bits are converted to $W^{(2)}$, $X^{(2)}$, $Y^{(2)}$, and $Z^{(2)}$. Traditionally, we can construct Boolean equations to represent this permutation. Then by simplifying those equations, we can tell the overhead of circuit implementation. Mathematically, this is a Multi-Output Logic Optimization Problem [17].

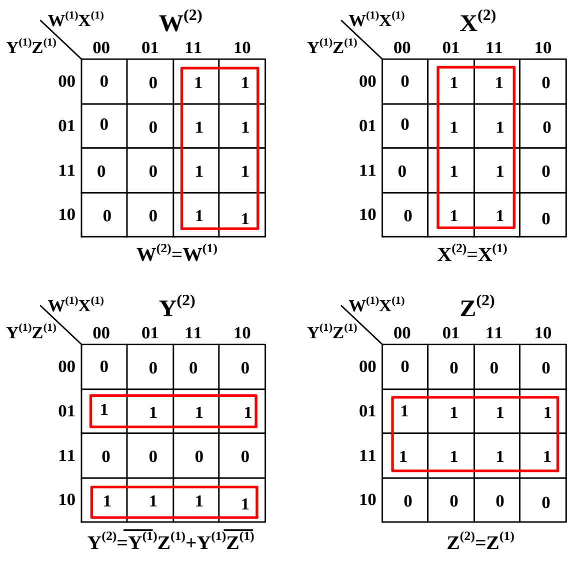


Fig. 6. Computing overhead by applying logic optimization

To solve this problem, as shown in Fig. 6, we draw Karnaugh Tables (one table to represent the conversion of one bit), and then do logic optimization in each table, finally arriving at the simplified Boolean equations that are listed beneath each table.

From those equations, we can see that *Permutation 2* only changes the value of Y bit, requiring one XOR gate to implement. Thus, the total overhead of *Permutation 2* is only one XOR gate (equal to 6 transistors [18]), while it can save one TCAM entry. In packet classification, one TCAM entry has 104 bits (though only 4 bits in this example). According to [19], one TCAM bit requires 20 transistors. It is easy to calculate that the overhead is much smaller than the resource saved. Actually, even if without TCAM, BP is still worth doing. For example, if directly synthesizing classifiers in Fig. 4, the Original Classifier needs four L-input gates while Classifier 2 needs only one (L-2)-input gate (L=104) and overhead is only two XOR gates.

3) *Processing Speed*. The system throughput is decided by the slower of preprocessing and TCAM searching. To ensure high performance, the preprocessing needs to be implemented by hardware.

4) *Programmability*. Because the classifier may require updates from time to time, programmability is another concern in the BP technique. Both FPGA and RAM memory can provide programmability for the preprocessing. But considering the requirement of processing speed and the complexity of storing a series of permutations into a memory, we suggest using FPGA.

5) *Power*. Because one typical reason for TCAM optimization is to reduce power consumption, we should guarantee that BP can save power. Due to its architectures, an SRAM-based FPGA is generally more power efficient than a TCAM with the same gate count. After applying BP, if the total gate count of FPGA and TCAM is smaller than the original gate count of TCAM, we can think power is saved. So to simplify the analysis, we consider only circuit size in the paper.

IV. BP Problem and complexity

We now formally define the block permutation problem as the following optimization problem.

BP Optimization Problem: For a given classifier C_1, suppose that P is the set of all possible series of permutations; find a series of permutations P_1 ($P_1 \in P$) to map C_1 to C_2, such that $|P_1| + |C_2|$ is minimized; i.e.,

$$arg \min_{P1 \in P}(|P_1| + |C_2|),$$

where $|P_1|$ is the FPGA size required by P_1, and $|C_2|$ is the TCAM size required by C_2. (We compare FPGA size and TCAM size on the basis of equivalent gate count; please refer to Section IX for the formulas.)

As we mentioned in the previous section, the computation of $|P_1|$ involves logic optimization, but the computation of $|C_1|$ is very straightforward. If a given classifier C_1 contains N M-bit rules, then $|C_1| = NM$ TCAM bits. It is easy to see that

$$arg \min_{P1 \in P}(|P_1| + |C_2|) \le |C_1|.$$

This is because the classifier will not be changed if we don't do any permutation. In this case, $P_1 = \emptyset$, $C_2 = C_1$, $|P_1| + |C_2| = |C_1|$. Thus, this optimization problem is equivalent to a series of decision problems as follows:

BP Decision Problem: For a given classifier C_1, suppose that P is the set of all possible series of permutations; check if there exists a series of permutations P_1 ($P_1 \in P$) to map C_1 to C_2, such that $|P_1| + |C_2| = k$ ($k = 1, \cdots, |C_1|$), where $|P_1|$ is the FPGA size required by P_1, and $|C_1|$ and $|C_2|$ are the TCAM sizes required by C_1 and C_2, respectively.

By trying k from 1 to $|C_1|$ to solve the decision problems, by no more than $|C_1|$ times, we can solve the optimization problem. However, each of these decision problems is very "hard" to solve. Even for a given series of permutations P, we cannot "quickly" verify the decision problem in "Polynomial-time", because the computation of $|P|$ requires logic optimization, which is known to be an *NP-hard* problem [20]. The complexity of logic optimization grows quickly as the number of dimensions grows. For example, *Quine-McClusky* algorithm [21] is a classic optimal solution for the logic optimization problem, but its run-time complexity is too high to support a large problem space.

To find the optimal solution for the BP problem, one possible way is brute-force. Such a solution, however, is unrealistic. Let us think about a brute-force method. As we know, block permutations only change rule distribution and don't add or delete any rule elements. No matter how many permutations we execute, the only difference between C_1 and C_2 is the positions of rule elements, so we can draw a mapping table to record the location changes of all rule elements. Actually, a mapping table represents a series of special permutations, in which each permutation only swaps two rule elements. By trying all possible mapping tables, we can get the optimal solution. If the number of dimensions is L (i.e., each rule has L bits), then the number of rule elements is 2^L. According to the mathematical theory of *Permutations and Combinations*, the number of mapping tables can be up to $(1 * 2 * \cdots * 2^L) = (2^L)!$. In packet classification, $L = 104$. We can see then that the search space is prohibitively huge. So brute-force is impractical.

Therefore, in this paper we develop a heuristic algorithm to efficiently search approximation solutions.

V. Compressing Classifiers

A. Terms and Concepts

Before introducing the heuristic BP algorithm, we first define several terms and concepts below:

1) *Block Size:* The size of a block is defined as the number of points that are contained in the block. To simplify the description, in this paper we use the number of wildcards in the Boolean representation to denote the size of a block. For example, we say the size of the block "0*1*" in *Table 2* of Fig. 4 is 2 wildcards.

2) *Distance:* The distance of two blocks is the Hamming distance of their Boolean representations. To get the distance, we can count the number of bits in which two Boolean representations have different non-wildcard values. For example, the distance between "0001" and "1101" is 2; the distance between "0*01" and "01*1" is 0.

3) *Direction*: Direction indicates how a block spans different dimensions in Boolean Space. We can judge the direction of a block by the positions of the wildcards in the Boolean representation. If two blocks have wildcards appearing exactly in the same bits of their Boolean representations, we say these two blocks are in the same direction. For example, "0*01" and "0*10" are in the same direction, while "0*01" and "*010" are not. Any two points that have no wildcard in their Boolean representations are considered in the same direction.

TABLE II
CONDITIONS OF MERGING AND PERMUTING TWO BLOCKS

Merging	Permutation	
	Target Blocks	Assistant Blocks
(1) Block distance = 1 (2) Same block size (3) Same direction (4) Same action	(1) Block distance >= 2 (2) Same block size (3) Same direction (4) Same action	(1) Block distance >= 1 (2) Same block size (3) Same direction (4) Cover one and only one target block

4) *Merging:* In Boolean Space, if two blocks meet the condition of "Merging" in Table II, we can directly merge them into one block. This operation is called *Merging*. Please note that "Same action" in Table II means all points in the related two blocks are associated with the same action.

5) *Permutation, Target Blocks, and Assistant Blocks:* A *Permutation* is specified by a pair of *Target Blocks* and a pair of *Assistant Blocks*. The operation of a permutation consists of two steps: swapping the assistant blocks, and then merging the target blocks. A pair of target blocks and its corresponding pair of assistant blocks should satisfy the conditions of "Target Blocks" and "Assistant Blocks" in Table II, respectively. According to the conditions, there should be one target block covered by one of the two assistant blocks and moved during the swapping, while the other target block remains fixed. This operation of swapping assistant blocks can reduce the distance between two target blocks to one, so that they can be merged. Consider *Table 1* of Fig. 4 as an example. We swap assistant blocks "**01" and "**11" to merge target blocks "0*01" and "0*10" (please note that "0*01" is covered by "**01"). Normally, to merge two target blocks, there might be more than one valid assistant block pair as options. For example, for target blocks "0*01" and "0*10" in *Table 1* of Fig. 4, there is another valid assistant block pair "**10" and "**11". Please note that the size of assistant blocks determines the overhead.

B. Properties

Before presenting the heuristic BP algorithm, we first introduce a series of properties of assistant blocks to narrow down the search space to reduce the computation complexity.

Property 1: If there are multiple pairs of candidate assistant blocks for a given pair of target blocks, to minimize the permutation overhead we should choose the largest assistant blocks to swap.

It is important to point out that swapping small blocks causes more overhead than swapping big blocks, because small blocks have fewer wildcards in the Boolean representations, hence involving more non-wildcard bits into the permutations. For example, in *Permutation 2* of Fig. 4, as we explained in Section III, we choose "**01" and "**11" as a pair of assistant blocks. As a result, the overhead is one XOR gate. But if we choose another pair of smaller assistant blocks, such as "0*01" and "0*11", we need one OR gate and two AND gates in addition to one XOR gate. Thus, to reduce the overhead, we should choose large blocks to swap when doing the permutation operations.

Here, we continue to introduce the following property that discloses the relation between assistant block size, target block size, and the distance of two target blocks. This property is very useful for reducing the computation complexity.

Property 2: Assuming that the size of an assistant block is W_p wildcards, the size of its corresponding target block is W_t wildcards, the distance between the two target blocks is D, and the dimension of Boolean Space is L (i.e., each rule contains L bits), there exists the following relationship:

$$W_t \leq W_p \leq (L - D)$$

We explain Property 2 using Lemmas 1-3 as follows:

Lemma 1: $W_p \geq W_t$.

Proof: Lemma 1 discloses the lower bound of assistant block size. According to Table II, there must be an assistant block fully covering a target block, because we need the former one to carry the latter one in the permutation to reduce the distance between the pair of two target blocks. For example, in *Table 1* of Fig. 4, the assistant block "**01" covers the target block "0*01", so the size of the assistant block cannot be less than the size of the target block. ■

Before introducing Lemmas 2 and 3, let us use Fig. 7 to illustrate how to find a pair of assistant blocks for a given pair of target blocks by deducing the Boolean representations. Suppose that there is a pair of target blocks *Target Block 1* and *Target Block 2*. Without loss of generality, we assume that these two target blocks have wildcards in the bits $W_1, \cdots, W_M$, the same values in the bits $X_1, \cdots, X_N$, and different values in the bits $Y_1, \cdots, Y_D$. According to the definition in Section V, their distance is D, and there should be one target block covered by one assistant block. Let us assume that it is *Target Block 1* covered by *Assistant Block 1*. Then, *Assistant Block 1* should have the same values as *Target Block 1* in $Y_1, \cdots, Y_D$ (we will prove later in Lemma 2 that these bits cannot be wildcards). The other bits of *Assistant Block 1* can be wildcards or the same values as *Target Block 1*. In the figure, they are all filled with wildcards. The next step is to deduce *Assistant Block 2*, the one to be swapped with *Assistant Block 1*. To merge the two target blocks, we need to reduce their distance to one, which requires the two assistant blocks to have same value in one bit among $Y_1, \cdots, Y_D$ and different values in the remaining $(D - 1)$ bits. In this figure, we assume that the two assistant blocks are the same in Y_D. Finally, we get a pair of assistant blocks.

Lemma 2: For a given pair of target blocks, if they have D different non-wildcard bits in the Boolean representations, then none of these D bits of their assistant blocks can be a wildcard.

Proof: This lemma is needed to prove Lemma 3. We prove it by contradiction, using the example in Fig. 7. If a wildcard appears in any bit of $Y_1, \cdots, Y_{D-1}$ of the assistant blocks, e.g., Y_1 of *Assistant Block 1(a)* and *Assistant Block 2(a)*, then Y_1 of

Target Block 1 will keep unchanged and this target block will be transformed to *Block 1(a)*. Because the distance of *Block 1(a)* and *Target Block 2* is not 1, we cannot directly merge them. If there is a wildcard in Y_D, as shown in *Assistant Block 1(b)* and *Assistant Block 2(b)*, then *Target Block 1* and *Target Block 2* will both be moved and turned to *Block 1(b)* and *Block 2(b)*, respectively. Obviously, *Block 1(b)* and *Block 2(b)* cannot be directly merged. So far, we have proved Lemma 2. ■

	L=M+N+D										
Bit Positions	W_1	···	W_M	X_1	···	X_N	Y_1	Y_2	···	Y_{D-1}	Y_D
Target Block 1	*	···	*	1	···	1	1	1	···	1	1
Target Block 2	*	···	*	1	···	1	0	0	···	0	0
Assistant Block 1	*	···	*	*	···	*	1	1	···	1	1
Assistant Block 2	*	···	*	*	···	*	0	0	···	0	1
Block 1(a)	*	···	*	1	···	1	1	0	···	0	1
Assistant Block 1(a)	*	···	*	*	···	*	*	1	···	1	1
Assistant Block 2(a)	*	···	*	*	···	*	*	0	···	0	1
Block 1(b)	*	···	*	1	···	1	0	0	···	0	1
Block 2(b)	*	···	*	1	···	1	1	1	···	1	0
Assistant Block 1(b)	*	···	*	*	···	*	1	1	···	1	*
Assistant Block 2(b)	*	···	*	*	···	*	0	0	···	0	*

Fig. 7. Scenarios of deducing Boolean representations of assistant blocks

Lemma 3: $W_p \leq (L - D)$.

Proof: Lemma 3 points out the upper bound of the assistant block size. Again, we use the target blocks in Fig. 7 as an example to prove this lemma. Based on Lemma 2, none of the bits $Y_1, \cdots, Y_D$ of the assistant blocks can be a wildcard. So the largest assistant blocks are those blocks whose $W_1, \cdots, W_M$ and $X_1, \cdots, X_N$ are all wildcards and $Y_1, \cdots, Y_D$ are all non-wildcards. Thus we know that the maximum assistant block size is $(M + N) = (L - D)$ wildcards. ■

Property 3: In packet classification, it holds that:

$$0 \leq W_p \leq (L - 2) = 102$$

Property 3 is an extension of Property 2. In packet classification, $\mathrm{L} = 104$; according to Table II, $\mathrm{D} \geq 2$. So, $\mathrm{W}_p \leq (L - D) \leq (L - 2) = 102$. In Boolean Space, a block should contain at least one rule element. Therefore, we have $\mathrm{W}_p \geq W_t \geq 0$ wildcards.

C. BP Algorithm

In this section, we propose the heuristic BP algorithm to compress classifiers. Our intention is to develop a practical algorithm with reduced computational complexity to find an approximation solution, by taking advantage of the aforementioned properties and lemmas and some predefined parameters. As shown in Fig. 8, the BP algorithm reads in a classifier and then recursively finds and performs permutations; after a predefined number of rounds have been completed, it will output a compressed classifier. The overall process consists of two phases: the direct logic optimization phase (Line 15) and the permutation phase (Lines 16-32).

In the direct logic optimization phase, we directly apply logic optimization on the original classifier to group adjacent rule elements. This is to reduce the number of rules that will be involved in the permutation phase and, hence, reduce the computation complexity.

```
1.  Function BP_CLASSIFIER_COMPRESS($C_0$,$N_r$,$W_{min}$,$N_{tp}$)
2.  Input:
3.          Original classifier $C_0$ in $L$-dimension Boolean Space;
4.          Number of rounds $N_r$;
5.          Minimum assistant block size $W_{min}$ (wildcards);
6.          Maximum number of target block pairs to be
        considered in each round $N_{tp}$;
7.  Output:
8.          Compressed classifier $C_1$;
9.  Constant:
10.         Maximum assistant block size $W_{max} = (L - 2)$;
11. Variable:
12.         A set of target block pairs $ST_1$;
13.         A pair of assistant blocks $A_1$;
14. Begin
15.   $C_1 = DIRECT\_LOGIC\_OPTIM(C_0)$;
16.   for $round = 0$ to $(N_r - 1)$ do
17.       for $W_p = W_{max}$ to $W_{min}$ do
18.           $ST_1 = \emptyset$;  $A_1 = \emptyset$;
19.           $ST_1 = FIND\_TARGET(C_1, W_p, N_{tp})$;
20.           if $ST_1 \neq \emptyset$ then
21.               $A_1 = FIND\_ASSISTANT(C_1, W_p, ST_1)$;
22.           end if
23.           if $A_1 \neq \emptyset$ then
24.               $C_1 = EXECUTE\_PERM(C_1, A_1)$;
25.               break;
26.           end if
27.       end for
28.       if $A_1 == \emptyset$ then
29.           return $C_1$;
30.       end if
31.   end for
32.   return $C_1$;
33. End
```

Fig. 8. Pseudo code of the BP classifier compression algorithm

In the permutation phase, we recursively find and perform permutations on the classifier. We use the parameter N_r to control the number of iteration rounds. According to the original idea of block permutation, we expect to find and execute only one permutation in each round of iteration. After the whole process is completed, we will have executed a series of permutations. Because a permutation requires a pair of target blocks and a pair of assistant blocks, the algorithm in each round uses three steps to find target blocks (Line 19) and assistant blocks (Lines 20-22), and then execute the permutation found (Lines 23-26). According to Property 1, a large assistant block leads to low overhead. So we always choose the largest possible assistant blocks. In each round of iteration, we start from the largest possible blocks, whose sizes are decided by W_{max} based on Property 3 (Line 10), to the smallest allowed blocks, whose sizes are decided by a predefined factor W_{min}. If we cannot find a permutation under the current constraint of the assistant block size, we will try a smaller size, until reaching W_{min}. The algorithm terminates when either of the following two conditions are met: (1) the algorithm has run for N_r rounds, or (2) the algorithm cannot find a valid pair of assistant blocks to swap in the current round. Next, we will explain the three functions that are called in each round of the permutation phase.

1) FIND_TARGET

```
1.  Function FIND_TARGET($C_1$, $W_p$, $N_{tp}$)
2.  Input:
3.          A Classifier $C_1$ with $N_1$ rules in $L$-dimension Boolean
        Space;
4.          Expected assistant block size $W_p$ (wildcards);
5.          Maximum number of target block pairs to be
        considered in each round $N_{tp}$;
6.  Output:
7.          A set of target block pairs $ST_1$;
8.  Begin
9.    $ST_1 = \emptyset$; M=0;
10.   for rule $i = 0$ to $(N_1 - 1)$ do
11.       for rule $j =$ i + 1 to $(N_1 - 1)$ do
12.           if Pair$(i, j)$ cannot be a pair of target blocks then
13.                 break;
14.           end if
15.           if the sizes of rule $i$ and j are larger than $W_p$ then
16.                 break;
17.           end if
18.           if the distance D$(i, j) \neq (L - W_p)$ then
19.                 break;
20.           end if
21.           $ST_1 = ST_1 +$ Pair$(i, j)$; M=M+1;
22.           if M $== N_{tp}$ then
23.               return $ST_1$;
24.           end if
25.       end for
26.   end for
27.   return $ST_1$;
28. End
```

Fig. 9. Pseudo code of the FIND_TARGET function

The purpose of the FIND_TARGET function is to find out all possible target block pairs based on the input parameters. As shown in Fig. 9, this function examines all rule pairs to check (1) if a rule pair meets the conditions of "Target Blocks" in Table II (Lines 12-14); (2) if their sizes satisfy Property 2 (i.e., not larger than W_p as shown in Lines 15-17); (3) whether their distance satisfies Property 2 (actually we need to do this only if the distance is equal to $(L - W_p)$ as shown in Lines 18-20 based on Lemma 4 which will be explained soon). Only if a pair of rules meets all these three constraints should we consider it as a pair of target blocks. These constraints can largely reduce the number of target block pairs that need to be considered in each round of iteration, hence reducing the computational complexity. We use the parameter N_{tp} to limit the number of target block pairs. If there are too many target blocks found, we report only the first N_{tp} pairs.

Lemma 4: In FIND_TARGET function, constraints $D = (L - W_p)$ and $D \leq (L - W_p)$ are equivalent in finding permutations. To reduce the computation complexity, we can consider only the target block pairs that satisfy $D = (L - W_p)$.

Proof: According to Property 2, we have $W_p \leq (L - D)$, which can be rephrased as D $\leq (L - W_p)$. Suppose that there are two pairs of target blocks t_{pair1} and t_{pair2} in the current input classifier C_1. Suppose the block distances in t_{pair1} and t_{pair2} are D_1 and D_2, respectively. Without loss of generality, we assume $D_1 < D_2$. Because we gradually decrease W_p to search target blocks, if we set the constraint as D $= (L - W_p)$, we will report t_{pair1} when W_p goes down to satisfy $W_p = (L - D_1)$; if we set the constraint as D $\leq (L - W_p)$, when $W_p = (L - D_1)$, we have $W_p > (L - D_2)$, which violates Lemma 2, so we cannot report t_{pair2}, only t_{pair1}. If we can find assistant blocks for t_{pair1}, we will execute a permutation and get a new classifier to run next round. If we cannot find assistant blocks for t_{pair1}, we will continue to decrease W_p and eventually report t_{pair2}, no matter if the constraint is D $= (L - W_p)$ or D $\leq (L - W_p)$. So far, whether the constraint is set to D $\leq (L - W_p)$ or D $= (L - W_p)$, we always get the same result. Hence, Lemma 4 is proved. ∎

2) FIND_ASSISTANT

If the target block set returned by the FIND_TARGET function is not empty, the BP algorithm will continue to run the FIND_ASSISTANT function to find the corresponding assistant block pairs. As shown in Fig. 10, FIND_ASSISTANT will find all possible assistant blocks whose size is equal to the input parameter W_p for each pair of target blocks (Lines 12-16). Then it will evaluate the compression effect of each pair of assistant blocks and choose only the one that can reduce the most number of rules (Line 17).

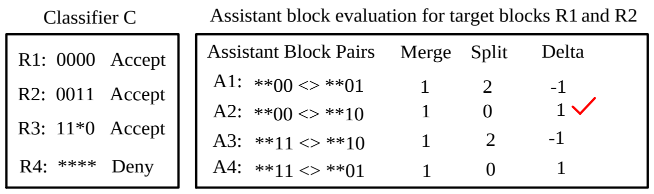
```
1.  Function FIND_ASSISTANT($C_1$, $W_p$, $ST_1$)
2.  Input:
3.          A Classifier $C_1$;
4.          Expected assistant block size $W_p$ (wildcards);
5.          A set of target block pairs $ST_1$;
6.  Output:
7.          A pair of assistant blocks $A_1$;
8.  Variable:
9.          Sets of Assistant block pairs $SA_1$ and $SA_2$;
10. Begin
11.   $SA_1 = \emptyset$;
12.   for each pair of target blocks $t_{pair} \in ST_1$ do
13.       $SA_2 = \emptyset$;
14.       $SA_2 =$ SUB_FIND_ASSIST$(C_1, W_p, t_{pair})$ ;
15.       $SA_1 = SA_1 + SA_2$;
16.   end for
17.    $A_1 =$ SUB_EVALUATE_ASSIST$(SA_1)$ ;
18.   return $A_1$;
19. End
```

Fig. 10. Pseudo code of the FIND_ASSISTANT function

The function of finding assistant blocks for a given pair of target blocks is implemented in the SUB_FIND_ASSIST sub-function (Line 14). Its main idea is to deduce the Boolean representations of assistant blocks from the Boolean representations of the given target blocks (this method has been shown in the proof of Lemma 2). According to Lemma 5, we can find $2 * (L - W_p)$ pairs of assistant blocks for a given target block pair.

Lemma 5: In the SUB_FIND_ASSIST sub-function, we can find exactly $2 * (L - W_p)$ pairs of assistant blocks for each given target block pair.

Proof: Without loss of generality, we still use the examples in Fig. 7 to prove this Lemma. When the distance of two target blocks is D bits, we need to inverse $(D - 1)$ bits among $Y_1, \cdots, Y_D$ of one of the target blocks to shorten their distance to 1. There are $2D$ possible operations. A pair of assistant blocks can be obtained in correspondence to each of the possible inversing operations. According to Lemma 4, $D = (L - W_p)$. Therefore, we can find exactly $2D = 2 * (L - W_p)$ pairs of assistant blocks. ■


Classifier C

R1:	0000	Accept
R2:	0011	Accept
R3:	11*0	Accept
R4:	****	Deny

Assistant block evaluation for target blocks R1 and R2

Assistant Block Pairs		Merge	Split	Delta
A1:	**00 <> **01	1	2	-1
A2:	**00 <> **10	1	0	1 ✓
A3:	**11 <> **10	1	2	-1
A4:	**11 <> **01	1	0	1

Fig. 11. Evaluating assistant blocks

The SUB_EVALUATE_ASSIST (Line 17) evaluates all the assistant block pairs and chooses the "best" one. There are two situations that we need to consider when swapping a pair of assistant blocks in a permutation. First, swapping a pair of assistant blocks may merge more than one pair of target blocks; thus a permutation can reduce multiple rules. For example, *Permutation 1* in Fig. 4 can reduce two rules. Second, a permutation may also break some existing blocks, leading to more rules. Thus we define the following metric *delta* to evaluate assistant blocks. We choose a pair of assistant blocks only if its *delta* is a positive number.

$$delta = \#\ of\ rules\ reduced\ - \#\ of\ rules\ created$$

The way to estimate the number of rules reduced for a given pair of assistant blocks is by checking all possible rule pairs in the current classifier to see if any of them can be a target block pair of the given assistant blocks, based on the conditions in Table II. To estimate the number of rules created, we scan all the rules in the current classifier to see if any wildcard in their Boolean representations would be affected by swapping the given assistant blocks. Fig. 11 is an example of evaluating assistant blocks. In this example, we list four pairs of assistant blocks for the target block pair $R1$ and $R2$. All these assistant blocks can merge $R1$ and $R2$, resulting in one rule reduced. But among these assistant blocks, $A1$ and $A3$ will split $R3$ and create two rules, while $A2$ and $A4$ will not. For example, $R3$ is made up of "1110" and "1100". If we choose $A1$ to swap, then "1100" will be changed to "1101", which cannot be merged with "1110", resulting in two new blocks while $R3$ disappears. Therefore, we choose $A2$ or $A4$.

3) EXECUTE_PERM

The function of EXECUTE_PERM (Line 24 in Fig. 8) is the last step of each round of the BP algorithm. This function will be called to execute a permutation if the previous step can return a pair of assistant blocks. To execute a permutation, the BP algorithm will first scan the current classifier to change the Boolean representations of the rules affected by swapping assistant blocks; second, it will compare rules; if any pair of rules meets the condition of "Merging" in Table II, then the algorithm will merge them into one rule.

Fig. 12 gives the details of how the BP algorithm works on the example in Fig. 4. In this example, the dimension of Boolean Space L is 4. Based on Property 3, we try assistant block size W_p from 2 wildcards to 0 wildcards in each round. The process is completed in two rounds. In the first round, when $W_p = 2$, we find four pairs of target blocks and sixteen pairs of assistant blocks (only four pairs of assistant blocks associated with the first target block pair are shown in the figure due to the limited space). These four assistant block pairs can provide the same *delta*. Among them, we randomly select one pair, say "$11 ** <> 01 **$", to execute the permutation and get the compressed *Classifier 1*. In the second round, when $W_p = 2$, we find only one pair of target blocks and four pairs of assistant blocks. Because all the four pairs of assistant blocks contribute to the same *delta*, we simply perform "$** 01 <> ** 11$" and then get *Classifier 2*, which is the final result.

D. Complexity

As we analyzed in Section IV, the Block Permutation problem is *NP-hard* and cannot be solved in polynomial time. The proposed BP algorithm can provide sub-optimal compression results with a relatively low run-time complexity.

On one hand, the BP algorithm can provide sub-optimal results because (1) it searches assistant blocks starting from the largest possible size $W_{max} = (L - 2)$ to make sure the permutation overhead is as low as possible; (2) it puts a cap on the minimum assistant block size W_{min} so that the overhead involved in each permutation can be upper bounded; (3) we define a metric called *delta* to make sure that each permutation can actually reduce the number of rules.

On the other hand, we limit the run-time complexity of the BP algorithm in three ways: (1) unlike brute-force, which does not consider rule distribution, the BP algorithm is sensitive to rule distribution. If the rule distribution is dense, a case in which it is unnecessary to apply the BP technique, the BP compression process will finish quickly; (2) we use a series of properties and lemmas to reduce the computation; (3) we also provide a method to manually control the run-time complexity by introducing the parameters of N_r and N_{tp}.

Now, let us estimate the worst case run-time complexity of the BP algorithm. Suppose that the classifier (after the direct logic optimization phase) contains N ternary string rules.

The worst case run-time of the BP algorithm in Fig. 8 is:

$$T_{BP} = \sum_{i=0}^{Nr-1} [(W_{max} - W_{min} + 1)(T_1(i) + T_2(i)) + T_3(i)]$$
$$< \sum_{i=0}^{Nr-1} [L(T_1(i) + T_2(i)) + T_3(i)]$$

Where, L is constant; $T_1(i)$, $T_2(i)$, and $T_3(i)$ is the worst case run-times of FIND_TARGET, FIND_ASSISTANT, and EXECUTE_PERM in the i^{th} round, respectively. Please note

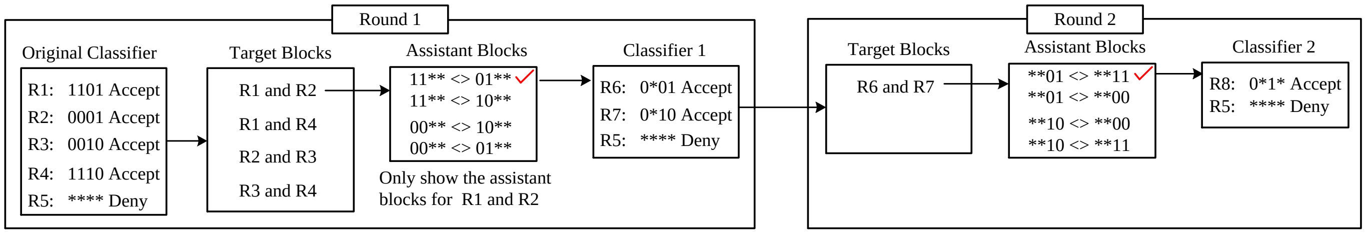


Fig. 12. A complete example of applying the BP algorithm

that EXECUTE_PERM is called only once because only one permutation is executed in each round.

In the worst case, each round can reduce only one rule, so in the i^{th} round, the number of rules is $N_1(i) = N - i$; the number of rule pairs is $N_2(i) = N_1(i)[N_1(i) + 1]/2$. Because L is constant, the run-time complexity of checking the Boolean representation of a rule can be considered a constant value. So, the worst case run-time of FIND_TARGET is:

$$T_1(i) = O(N_2(i))$$

For FIND_ASSISTANT, based on Lemma 5, the maximum number of assistant block pairs found is $N_3(i) = 2(L - W_p)N_{tp} < 2LN_{tp}$. From the proof of Lemma 5, we can also know that the run-time of Lines 12-16 in Fig. 10 is $T_{21}(i) < 2LN_{tp}$; and for SUB_EVALUATE_ASSIST in Line 17, its run-time $T_{22}(i) = O(N_2(i))N_3(\mathrm{i}) + O(N_1(i))N_3(i)$. So, the worst case run-time complexity of FIND_ASSISTANT is:

$$T_2(i) = T_{21}(i) + T_{22}(i) < (O(N_1(\mathrm{i}) + N_2(\mathrm{i})) + 1)2LN_{tp}$$

The worst case run-time of EXECUTE_PERM is:

$$T_3(i) = O(N_1(i)) + O(N_2(i)) = O(N_1(\mathrm{i}) + N_2(\mathrm{i}))$$

Based on all the analyses, we can finally deduce that the worst case run-time of BP algorithm is

$$T_{BP} = O(N_r N_{tp} N^2)$$

This means that once the N_r and N_{tp} have been decided, the worst case run-time complexity of BP algorithm is $O(N^2)$. Though the complexity has been reduced to "polynomial-time", in some cases the run-time may be still too long due to the large coefficients, especially when $L = 104$. One strategy is to reduce parameters like N_r and N_{tp}, but this may sacrifice the compression performance. So, a good tradeoff between run-time and compression is needed in real applications. When N becomes larger, because of the large coefficients, the rum-time may still grow quickly. For large classifiers, we can consider a method called *Classifier Partitioning* to reduce the run-time complexity by partitioning a large classifier into several small parts and then applying BP to each part. Based on Cauchy's Inequality, our preliminary suggestion is to partition the classifier as evenly as possible to get the minimum total run-time of all sets. After partitioning, while run-time is reduced, we can get a better compression. This is because if we keep the same N_{tp} on the original classifier for all parts, in each round, we can actually consider more target block pairs in total. We do not suggest partitioning a classifier to very small parts, because this would lessen the chance of finding target blocks. The parts can be processed in parallel to further save time.

VI. Implementing Permutations

As we have explained, when the classifier is compressed by a series of permutations, correspondingly we need to apply the same permutations to the incoming packet headers. Circuit size and throughput performance are the two major performance metrics we have to consider when implementing the permutations. In this section, we first introduce the *basic methodology* of designing the permutation logic circuit without considering throughput performance, and then propose a scheme called *stage-grouping* to achieve a tradeoff between circuit size and throughput.

A. Basic Methodology

If not considering performance, we can design an optimized circuit (i.e., with minimal size), by deducing and simplifying the final Boolean equations for a series of permutations.

Permutation (1)	**Permutation (2)**	**Final Equation**
11<>01****	****01<>**11**	**Substitute (1) into (2)**
$W^{(1)} = \overline{W}X + W\overline{X}$	$W^{(2)} = W^{(1)}$	$W^{(2)} = \overline{W}X + W\overline{X}$
$X^{(1)} = X$	$X^{(2)} = X^{(1)}$	$X^{(2)} = X$
$Y^{(1)} = Y$	$Y^{(2)} = \overline{Y^{(1)}}Z^{(1)} + Y^{(1)}\overline{Z^{(1)}}$	$Y^{(2)} = \overline{Y}Z + Y\overline{Z}$
$Z^{(1)} = Z$	$Z^{(2)} = Z^{(1)}$	$Z^{(2)} = Z$

Fig. 13. Boolean equations of the two permutations in Fig. 4

Fig. 13 provides the Boolean equations of the two permutations used in Fig. 4. We can see that each equation in a permutation can always be implemented by one XOR gate. Given a packet header, we can calculate the transformed values of W, X, Y, and Z bits by using the two equations in series. We can also substitute the equations of *Permutation (1)* into those of *Permutation (2)* to get a single set of final equations, with which we can directly calculate the transformed value of the packet header. For a general case, there is a method to easily deduce the final equations. Let us suppose that in a permutation, we swap two assistant blocks that have same values in bit positions $X_1, \cdots, X_m$, have different values in bit positions $Y_1, \cdots, Y_n$, and have wildcards in other bit positions. And we denote their Boolean representations as"$a_1 \cdots a_m b_1 \cdots b_n * \cdots *$ " and "$a_1 \cdots a_m \overline{b_1} \cdots \overline{b_n} * \cdots *$ " ("$\overline{b}$" is the inverse of "b"). In the permutation, only the $Y_1, \cdots, Y_n$ bits of the incoming packet header will be changed. Assuming that the value of the $X_1, \cdots, X_m, Y_1, \cdots, Y_n$ bits of the incoming packet header is $c_1, \cdots, c_m, d_1, \cdots, d_n$ respectively, and after the permutation, their $Y_1, \cdots, Y_n$ bits will be changed to $d'_1, \cdots, d'_n$, respectively, we use the following Boolean equations to calculate the new values of $Y_1, \cdots, Y_n$ bits.

$$\begin{cases} d_1' = \overline{d_1} \cdot F + d_1 \cdot \overline{F} \\ \vdots \\ \vdots \\ d_n' = \overline{d_n} \cdot F + d_n \cdot \overline{F} \end{cases}$$

Where, $F = 1$ if $c_1 \cdots c_m = a_1 \cdots a_m$ and $d_1 \cdots d_n = b_1 \cdots b_n$ *or* $\overline{b_1} \cdots \overline{b_n}$; Otherwise, $F = 0$.

B. *Stage-Grouping Methodology*

For a given series of permutations, intuitively, we can use the *pipeline structure* to implement them in circuits. If there are N permutations, we can design an N*-stage* pipeline, with each stage implementing one permutation. A packet needs to traverse N stages with a delay of N clock cycles before entering the TCAM for the classification. Because each stage is simple enough, the pipeline can run at a high clock rate and thus provide a high throughput. One downside of using this pipeline stage is that it usually requires large hardware resources.

An alternative solution to the pipeline is to use a combinational logic to implement all N permutations. This structure is a *1-stage* pipeline solution, which is actually the same as the *basic methodology* that we have explained. Normally, a *1-stage* pipeline requires much less hardware resources than an N*-stage* pipeline, because we can simplify the Boolean equations. However, the relatively high critical path delay, which would lower the clock rate, is a major concern when we use the *1-stage* solution.

```
1.  Methodology Stage-Grouping($P_1, P_1, \cdots, P_N, R_t, S_{max}$)
2.  Input:
3.          A series of permutations $P_1, P_1, \cdots, P_N$;
4.          Targeted clock rate $R_t$;
5.          Maximum circuit size $S_{max}$;
6.  Output:
7.          $M$-stage pipeline $PL$;
8.  Variable:
9.          Current pipeline $PL_0$, $PL_1$;
10.         Current clock rate $R$;
11.         Current circuit size $S$;
12. Begin
13.  $M = 0$; $S = 0$; $i = 1$; $PL = \emptyset$;
14.  while ($i \le N$ && $S < S_{max}$) do
15.     $PL_0 = PL$;
16.     for ( $j = i; j \le N; j = j + 1$) do
17.           Group permutations from $P_i$ to $P_j$ into a new stage,
        append this stage into $PL_0$ to form $PL_1$;
18.           Synthesize $PL_1$, and get $R$, $S$;
19.           if ($R > R_t$ && $S < S_{max}$) then
20.                $PL = PL_1$;
21.           else $M = M + 1$; $i = j$; break;
22.           end if
23.     end for
24.  end while
25.  return $M$-stage pipeline $PL$;
26. End
```

Fig. 14. The methodology of stage-grouping

Considering the pros and cons of both *1-stage* and N*-stage* structures, we hereby propose a solution called *stage-grouping,* which is able to find the best number of stages to achieve a tradeoff between cost and speed. This method is to group consecutive pipeline stages together to reduce the number of pipeline stages. Each new stage implements multiple permutations, and the new Boolean function of each new stage can be derived by using the *basic methodology*.

Fig. 14 shows the proposed solution of *stage-grouping*. The *stage-grouping* methodology constructs a pipeline by adding stages one by one. Based on greedy strategy, each stage is generated by grouping as many permutations as possible, as long as the targeted clock rate can be satisfied.

Because the overall throughput of a pipeline is determined by the slowest stage(s), it is possible that some exceptionally complicated permutations would slow down the pipeline. To address this problem, a simple idea is to duplicate the bottleneck stage(s). Please note that only the bottleneck stage(s) rather than the whole pipeline need to be duplicated. So the extra overhead can be small while the throughput is improved.

Another concern is that the implementation in reality would be limited by the capacity of FPGA built in the hardware system. So, we add a parameter S_{max} in the *stage-grouping* methodology to limit the size of the pipeline. It is possible that $S > S_{max}$ before i reaches N (Line 14). As a result, only part of the permutations can be put into FPGA. In this case, we should store the corresponding intermediate classifier rather than the final classifier of BP compression into TCAM.

VII. DISCUSSION

In this section, we present the architecture in Fig. 15 for discussion, which is modified from that in Fig. 5 to support classifier updates and partitioning. As we have mentioned in previous sections, for a large classifier, we can partition it to several small parts and then apply BP on each individual part. Each partition requires one FPGA and one TCAM to implement. As for classifier incremental updates (e.g., inserting or deleting some rules), we use a small TCAM called *Scratch TCAM* with high priority to store scratch entries representing the difference between the new classifier and old classifier. As shown in Fig. 15, a small component called *Priority Control* is used to choose which action to be used. If a packet matches both the scratch TCAM and the partition(s), the action from the scratch TCAM will be finally reported.

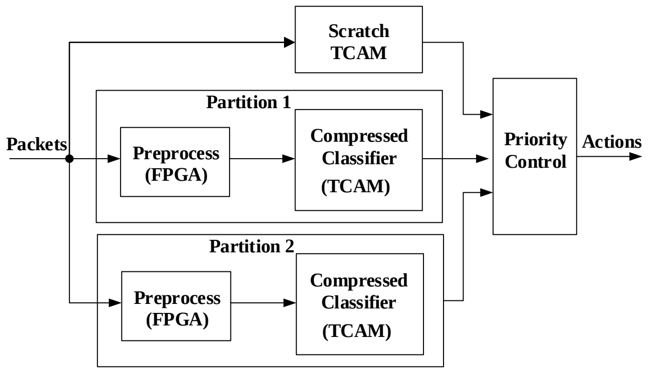


Fig. 15. Modified architecture for classifier update and partitioning

When the system starts, the scratch TCAM is empty. To insert or delete rules, we compare the Karnaugh Tables of the new and old classifiers and generate the delta Karnaugh Table, inside which some points are changed to “Accept” or “Deny” and marked with “AC” or “DC”, and other points are not changed and marked with "ANC" or "DNC". Then by applying logic optimization to the "AC" points and "DC" points separately, we can get an optimized number of "AC" and "DC" rules (no default rule), which are then stored in the scratch TCAM. Note that the "AC" (or "DC") points can merge some "ANC" (or "DNC") points to get a better optimization. The run time of logic optimization on the small delta Karnaugh Table is usually tiny. If an incremental update is required when the system is still active, two scratch TCAMs can be used alternatively. Because the scratch TCAM is relatively small, the duplication will not cost much. Note that the scratch TCAM will be searched directly using the original packet headers without permutation. When there are enough updates accumulated, we can do a complete BP compression to move everything into the main TCAMs and free up the scratch TCAM. Considering that BP compression and FPGA synthesis take time, we should start a new complete BP process earlier, before scratch TCAM becomes full.

VIII. Experiments

A. *Experiment Setting*

Our experiments were based on one real-life firewall classifier and several artificial classifiers generated by using ClassBench [15]. We generated three typical types of artificial classifiers: Firewall (FW), Access Control List (ACL), and IP Chain (IPC). As Table III shows, the first eight classifiers vary in size from 60 to 660 rules. The average prefix expansion ratio of these classifiers is 1.91. The largest expansion is observed in classifier ipc-1, whose expansion ratio is 2.89, where 202 rules are expanded to 584 ternary strings (or TCAM entries). We specifically added two large classifiers. One is acl-4 containing 1209 rules and expanding to 1725 TCAM entries. The other is acl-5 containing 3708 rules and expanding to 4880 TCAM entries. We partitioned acl-4 to two parts and acl-5 to four parts to test the performance of classifier partitioning. For convenience, we just tried to make the number of entries of the parts close after range expansion. A better partitioning should consider the entries after logic optimization.

To evaluate the performance, we compared the BP technique with McGeer's algorithm [14], which is the first bit-level scheme. As shown in Fig. 2, McGeer's algorithm uses direct logic optimization (called Heuristic 1) and a process based on the first-matching property (called Heuristic 2) to reduce entries. BP does not use the first-matching property, so it can be combined with McGeer's algorithm. For each classifier, we first applied direct logic optimization, then we applied (1) Heuristic 2; and (2) the BP compression algorithm followed by Heuristic 2. Because Heuristic 2 would change the classifier style, as required by Espresso [22] and the BP program, that all rules are "Accept" except for the default rule as "Deny" (please note that a classifier will be changed to this style after being mapped to Boolean Space), we have to do it as the last step. We chose a sub-optimal solution, Espresso, for logic optimization, since it has been identified as an *NP-hard* problem [20].

We implemented the BP algorithm using C++ language and performed the compression experiments on our Linux workstation driven by *Intel Xeon 2.0GHz E5335* CPUs. In packet classification, Boolean Space dimension L is 104. Parameters were set to $N_r = 150$ and $W_{min} = 54$. N_{tp} was set to 3 for the first eight classifiers and to 1 for acl-4, acl-5 and their partitions. All the partitions were processed in parallel by multiple CPU Cores. In Table III, we provide the longest run-time of the partitions. With these data, we are also able to estimate the total run-time. According to the suggestion in [14], if McGeer's Heuristic 2 cannot finish in 12 hours, we just stop it. Our targeted throughput was set to 100M packets per second. Based on that, we selected *Altera Cyclone III* FPGA [23]. We designed scripts to automatically generate Verilog codes based on the compression results, then synthesized them by using *Quartus* [23] on a Dell D630 laptop computer.

B. *Classifier Compression*

In Table III, we present the experimental results of direct logic optimization, BP algorithm, and McGeer's Heuristic 2. For the first eight classifiers, the BP compression process can reduce entries by 31.88% on average in addition to the 22.12% contributed by logic optimization; Applying McGeer's Heuristic 2 over "BP output" can reduce 1.29% entries and over "logic optimization output" can reduce 7.86%.

In the IPC classifiers, while block permutation can save 53.93% of entries on average, direct logic optimization can barely give any compression. Especially in ipc-2, the compression of direct logic optimization is 0. The reason for this low compression rate in the direct logic optimization phase is that the rule distributions of the IPC classifiers are very "sparse", so direct logic optimization can barely merge rules. This is what motivated our research on the BP technique.

In the ACL classifiers, we always find that block permutation contributes much more compression than direct logic optimization does, a fact from which we can judge that the ACL classifiers also fall into "sparse" rule distributions.

In the FW classifiers, we witness a compression ratio of 61.23% on average by direct logic optimization. In these cases, the average compression ratio of block permutation is 6.84%, which is much smaller than that of direct logic optimization. The reason is that the rule distributions of the FW classifiers are quite "dense", so direct logic optimization has good performance. In the real-life classifier *real-1*, because the classifier is closer to "dense" rule distribution than "sparse" rule distribution, direct logic optimization contributes a larger compression ratio than block permutation does. In this case, however, block permutation can still reduce 83 entries.

While bit-level schemes can yield good compression, run-time is a challenge. For example, from Table III we can see that McGeer's Heuristic 2 takes hours to run. In many cases, it cannot generate any outcome within 12 hours. As for BP, we can make a tradeoff between run-time and compression performance. For the first eight classifiers, if we set N_{tp} to 1, the average run-time can be only 1.9 minutes, but the compression ratio would drop sharply to 21.13%. In contrast, we found that $N_{tp} = 3$ is a better tradeoff, with which the average run-time is 3.4 minutes and the compression is 31.88%. But for acl-4 and acl-5, with $N_{tp} = 3$, the run-time is too long.

TABLE III
CLASSIFIER STATISTICS AND RESULTS FROM BP COMPRESSION EXPERIMENTS

Source	Classi-fier	# of Rules	# of Entries	Expan Ratio	Logic Optimization (LO)			Block Permutation (BP)				Heuristic 2 over BP			Heuristic 2 over LO		
					entries reduced	compr ratio	runtime (sec)	entries reduced	compr ratio	runtime (min)	# of perms	entries reduced	compr ratio	runtime (hour)	entries reduced	compr ratio	runtime (hour)
Class-Bench	fw-1	60	115	1.92	69	60.00%	0.33	12	10.43%	0.001	7	0	>0.00%	>12	0	>0.00%	>12
	fw-2	132	277	2.10	173	62.45%	34.72	9	3.25%	0.02	9	0	>0.00%	>12	0	>0.00%	>12
	acl-1	187	357	1.91	50	14.01%	0.18	146	40.90%	2.57	80	10	2.80%	2.71	57	15.97%	4.76
	acl-2	217	271	1.25	1	0.37%	0.18	137	50.55%	0.84	125	0	0.00%	1.02	0	0.00%	2.02
	acl-3	221	312	1.41	3	0.96%	0.56	99	31.73%	5.03	83	0	0.00%	10.58	0	>0.00%	>12
	ipc-1	202	584	2.89	14	2.40%	0.39	289	49.49%	5.68	89	31	5.31%	4.59	111	>19.01%	>12
	ipc-2	207	538	2.60	0	0.00%	0.26	314	58.36%	4.34	106	12	2.23%	1.54	150	27.88%	3.74
Real-life	real-1	660	802	1.22	295	36.78%	1.45	83	10.35%	5.83	50	0	>0.00%	>12	0	>0.00%	>12
	avg	235.75	407	1.91	75.63	22.12%	4.76	136.13	31.88%	3.04		6.63	1.29%		39.75	7.86%	
Class-Bench	acl-4	1209	1725	1.43	51	2.96%	2.18	183	10.61%	88.69	150	0	>0.00%	>12	0	>0.00%	>12
	part-1	750	829	1.11	16	1.93%	0.84	188	22.68%	8.53	150	0	0.00%	8.7	0	>0.00%	>12
	part-2	459	896	1.95	26	2.90%	0.79	306	34.15%	18.52	150	31	>3.46%	>12	50	>5.58%	>12
	2 parts in total	1209	1725		42	2.43%	0.84	494	28.64%	18.52		31	>1.80%	>12	50	>2.90%	>12
Class-Bench	acl-5	3708	4880	1.32	106	2.17%	3.27	266	5.45%	91.48	150	11	>0.23%	>12	0	>0.00%	>12
	4 parts in total	3708	4880		68	1.39%	1.12	1030	21.11%	21.47		121	>2.48%	>12	188	>3.85%	>12

TABLE IV
RESULTS FROM FPGA IMPLEMENTATION EXPERIMENTS

Classifier		TCAM saved by Logic Optimization		TCAM saved by Block Permutation		FPGA consumed			Ratio-1	Ratio-2	Pipeline	
		Entries	Gate Count	Entries	Gate Count	CFs	Registers	Gate Count			Stages	Clock Rate
fw-1		69	35880	12	6240	68	104	828	13.27%	1.97%	1	135.85
fw-2		173	89960	9	4680	51	104	777	16.60%	0.82%	1	127.57
acl-1		50	26000	146	75920	2498	2184	20598	27.13%	20.21%	21	108.17
acl-2		1	520	137	71240	3556	1664	20652	28.99%	28.78%	16	107.68
acl-3		3	1560	99	51480	2670	2288	21738	42.23%	40.98%	22	103.57
ipc-1		14	7280	289	150280	2605	2600	23415	15.58%	14.86%	25	106.37
ipc-2		0	0	314	163280	3098	2184	22398	13.72%	13.72%	21	100.94
real-1		295	153400	83	43160	1372	1456	12852	29.78%	6.54%	14	114.65
acl-4	part-1	16	8320	188	97760	2462	2808	24234	24.79%	22.85%	27	100.55
	part-2	26	13520	306	159120	4509	4368	39735	24.97%	23.02%	42	112.11
acl-5	4 parts on avg.	17	8840	285	148200	3661	3328	30951	20.88%	19.71%	32	106.23
avg.		60.36	31389.09	169.82	88305.45	2413.64	2098.91	19834.36	23.45%	17.59%	20.18	111.24

So we set it to 1 at the cost of lower compression performance. In the experiments, we found that classifier partitioning is very effective in handling large classifiers. After partitioning, with the same $N_{tp} = 1$, the run-time of acl-4 can be further reduced to 18.52 minutes and the compression also be improved to 28.64%. We also observed the similar effect on acl-5. This is because after partitioning, we can consider more target block pairs (one from each partition) in each round, as opposed to one pair considered by the original compression scheme.

C. FPGA Implementation

In this section, we discuss our experiments on FPGA implementation. In the experiments, we evaluated the overhead of the BP technique, which covers two aspects: hardware cost and operation performance of packet classification. The results are presented in Table IV.

For hardware cost, we used the concept of "Equivalent Gate Count" to estimate the actual hardware resource saved by using the BP technique (TCAM resource reduced minus FPGA resource consumed). From the TCAM chip ICFWTNM1 [19], we can estimate that the implementation of one TCAM bit requires about 20 transistors. Because a standard 2-input NAND gate consists of 4 transistors, we have the following equation:

$$TCAM\ Gate\ Count = \frac{\#\ of\ entries \times 104\ bits \times 20\ transistors}{4\ transistors}$$

The Altera FPGA resource consumption is reported in Combinational Functions (CFs) and Registers. In the experiments, we calculate the FPGA gate count as follows:

$$FPGA\ Gate\ Count = \#\ of\ CFs\ \times 3 + \#\ of\ Registers\ \times 6$$

The throughput requirement of the packet classification operation was set to no less than 100M packets per second. Accordingly, the clock rate of the pipeline should be no less than 100MHz. As Table IV shows, on average we need to use around 20 pipeline stages to meet the timing requirement, and the actual average clock rate estimated is 111.24MHz, while the fastest clock rate is 135.85MHz. Based on this performance, the average gate count of FPGA consumption is only 23.45% of that of the TCAM resource saved in the permutation phase (please see *Ratio-1* in Table IV). For a more accurate analysis, the TCAM saved in the direct logic optimization phase should be included, and then the average ratio of the FPGA overhead to the total TCAM saved by both direct logic optimization and block permutation can be as low as 17.59% (please see *Ratio-2* in Table IV). The FPGA overhead of the ACL classifiers is relatively high when compared to the TCAM saved. This is because the compressions are achieved by swapping relatively small permutations of blocks. To improve throughput, normally, we can use more stages. By this way, we can make each stage smaller and thus run at a higher clock rate. But the overall hardware cost will increase.

Our way of implementing FPGA is to use the *stage-grouping* methodology in Fig. 14. During the implementation, we tried to pack as many permutations into one stage as possible before constructing the next one. As Line 20 in Fig. 14 indicates, we should add one permutation at a time to a stage. But in the experiments, to reduce the synthesis time, we have tried to add multiple permutations each time. On average, the implementation time is 42.39 minutes (the FPGA experiments were done on a laptop computer; the implementation time can be shorter if using a high performance computer).

So far, in our experiments, we have shown that the proposed BP technique can significantly reduce TCAM entries while the overhead is much smaller than the resource saved.

IX. CONCLUSION

In this paper, we propose a new technique called Block Permutation (BP) to reduce the number of TCAM entries required to represent a classifier. The BP technique significantly improves the compression rate under the circumstances where direct logic optimization cannot perform effectively. The improvement is achieved by performing a series of permutations to change the distribution of rule elements in Boolean Space from sparse to dense, thus allowing more rules to be merged into each TCAM entry. The proposed BP is a new technique in that it searches for nonequivalent classifiers rather than equivalent ones, as previous schemes did. Because BP is a technique related to the common topic of logic optimization, it is not limited to the applications of packet classification and TCAM, but can also be applied to other hardware implementation-based applications.

REFERENCES

[1] D.E. Taylor, “Survey and taxonomy of packet classification techniques,” *ACM Computer Surverys*, pp. 238–275, 2005.

[2] Y. Xu, Z. Liu, Z. Zhang, H. J. Chao, “An Ultra High Throughput and Memory Efficient Pipeline Architecture for Multi-Match Packet Classification without TCAMs”, *ACM/IEEE ANCS*, 2009.

[3] K. Pagiamtzis and A. Sheikholeslami, “Content-addressable memory (CAM) circuits and architectures: A tutorial and survey,” *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar 2006.

[4] E. Spitznagel, D. Taylor, and J. Turner, “Packet classification using extended tcams,” *IEEE ICNP*, 2003.

[5] A. Bremler-Barr and D. Hendler, "Space-Efficient TCAM-based Classification Using Gray Coding," in *IEEE INFOCOM*, 2007.

[6] A. Bremler-Barr, D. Hay and D. Hendler, "Layered Interval Codes for TCAM-based Classification," in *IEEE INFOCOM*, 2009.

[7] M. Bando, N. S. Artan, R. Wei, X. Guo and H. J. Chao, "Range Hash for Regular Expression Pre-Filtering," *ACM/IEEE ANCS*, 2010.

[8] C. R. Meiners, A. X. Liu and E. Torng, “Topological Transformation Approaches to Optimizing TCAM-Based Packet Classification Systems,” in *ACM SIGMETRICS*, 2009.

[9] O. Rottenstreich and I. Keslassy, “Worst-Case TCAM Rule Expansion,” in *IEEE INFOCOM*, 2010.

[10] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, “Compressing forwarding tables,” in *IEEE INFOCOM*, 2013

[11] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, “Compressing Forwarding Tables for Datacenter Scalability,” in *IEEE JSAC Switching and Routing for Scalable and Energy-efficient Datacenter Networks* 2014..

[12] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, “Packet classifiers in ternary CAMs can be smaller,” in *SIGMETRICS*, 2006.

[13] C. Meiners, A. X. Liu, and Eric Torng, “TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs,” in Proceedings of *IEEE ICNP*, 2007.

[14] R. McGeer and P. Yalagandula, “Minimizing Rulesets for TCAM Implementation,” in Proceedings of *IEEE INFOCOM*, 2009.

[15] D. E. Taylor and J. S. Turner, “ClassBench: A Packet Classification Benchmark,” *IEEE INFOCOM*, 2005

[16] Maurice Karnaugh, “The Map Method for Synthesis of Combinational Logic Circuits,” *Transactions of the American Institute of Electrical Engineers* part I 72 (9): 593–599, 1953.

[17] G. D. Hachtel and F. Somenzi, “Logic synthesis and verification algorithms,” *Kluwer Academic Publishers*, 2002.

[18] J. M. Rabaey, “Digital Integrated Circuits (2nd Edition),” chap. 6, *Prentice-Hall*, 2003

[19] University of Waterloo, IC Tape-out History, http://www.ece.uwaterloo.ca/~cdr/www/chip.html

[20] C. Umans, “Complexity of two-level logic minimization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2006

[21] V.P. Nelson, “Digital Circuit Analysis and Design,” Prentice Hall, 1995

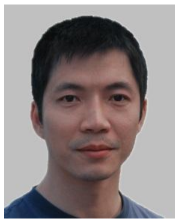
[22] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli, “Espresso-signature: A new exact minimizer for logic functions,” *IEEE Transactions on VLSI Systems*, 1993.

[23] Altera Cyclone FPGA and Quartus Tool. http://www.altera.com/

[24] C. Meiners, A. X. Liu, and Eric Torng, “Bit Weaving: A Non-prefix Approach to Compressing Packet Classifiers in TCAMs,” in Proceedings of *IEEE ICNP*, 2009.

[25] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan and E. Porat, "On Finding an Optimal TCAM Encoding Scheme for Packet Classification," in *IEEE INFOCOM*, 2013.

[26] R. Wei, Y. Xu, and H. J. Chao, “Block permutations in Boolean Space to Minimize TCAM for Packet Classification,” in *IEEE INFOCOM*, 2012.



Rihua Wei (M’12) received his B.S. degree in Telecom Engineering from Beijing University of Posts and Telecoms, China, in 1999, and his M.S. degrees in electrical and electronics engineering from Tsinghua University, China, in 2003, and from New York University Polytechnic School of Engineering, USA, in 2011. He is currently with Tensorcom Inc., Carlsbad, CA, USA. His areas of interests include networking, ASIC and System-on-Chip.

Yang Xu (M’05) received his B.E. degree from Beijing University of Posts and Telecoms, China, in 2001, and his PhD in computer science and technology from Tsinghua University, China, in 2007. He is currently a Research Associate Professor with the Department of Electrical and Computer Engineering, New York University Polytechnic School of Engineering, USA. His research interests include software-defined networks, data center networks, network on chip and high-speed network security.

H. Jonathan Chao (M’83-F’01) received his PhD in EE from The Ohio State University in 1985. He is currently a Professor with the Department of Electrical and Computer Engineering, New York University Polytechnic School of Engineering, Brooklyn, NY, USA, where he joined the faculty in January 1992. He has been doing research in the areas of network designs in software defined networking, datacenters, terabit switches/routers, network security and network on the chip.