Block Permutations in Boolean Space to Minimize TCAM for Packet Classification

Rihua Wei, Yang Xu and H. Jonathan Chao
Department of ECE, Polytechnic Institute of New York University
rwei01@students.poly.edu, {yangxu, chao}@poly.edu

***Abstract*—Packet classification is one of the major challenges 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, TCAM-based packet classification has the well-known range expansion problem, resulting in a huge waste of TCAM entries. In this paper, we propose a novel technique called *Block Permutation (BP)* to compress the packet classification rules stored in TCAMs. The compression is achieved by performing block-based permutations on the rules represented in Boolean Space. We develop an efficient heuristic approach to find the permutations for compression and design its hardware implementation. Experiments on ClassBench classifiers and ISP classifiers show that the proposed BP technique can reduce TCAM entries by 53.99% on average.**

Keywords-Packet Classification; TCAM; Range Expansion; Classifier Minimization; Logic Optimization

I. INTRODUCTION

Packet classification has been 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.

Table 1 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	Deny
r2	166.111.*.*	192.168.1.*	*	*	*	Accept
r3	*	*	*	*	*	Deny

Table 1 gives a sample classifier with three rules, in which each rule specifies a pattern with five fields (i.e., source IP and destination IP (prefixes), source port and destination port (ranges), and protocol type). From the geometric point of view, each rule can be viewed as a hyper-rectangle (also called a block) in the 104-dimensional Boolean Space corresponding to the 104 bits in the five fields.

Ternary Content Addressable Memories (TCAMs) have been 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 in either '0', '1', or '*' (don't-care). Before a rule can be stored in TCAMs, its range fields have to be converted to prefixes. For example, Rule *r2* in Table 1 requires only one TCAM entry since it contains only prefix fields. But for Rule *r1*, both the source port and destination port contain a range [1, 5]. So both of them needs to be expanded to three prefixes, i.e., "001", "01*", "10*". The combination of the prefix specifications of the two ranges will consume $3 \times 3 = 9$ TCAM entries, causing the well-known *range expansion* problem[1]. Because TCAMs are expensive and power-hungry, the range expansion problem increases the already high implementation cost of TCAMs.

Thus, it is very important to reduce the TCAM entries that are required to represent a classifier. Previous work in this field can be classified into three categories: *TCAM Hardware Improvement* [4], *Range Encoding* [5][6][7][12][13] and *Classifier Compression* [8][9][10][11][14]. In this paper, we propose a new classifier compression technique called *Block Permutation (BP)*, which is motivated by our observation that the existing schemes perform badly under some circumstances. Our contributions are summarized as follows:

(1) The existing classifier compression schemes normally find semantically equivalent but smaller TCAM representations for the packet classifiers. In contrast, **the BP technique reduces TCAM entries by converting the original classifiers to a smaller space but unnecessarily equivalent to the TCAM representations.**

(2) We propose an efficient heuristic approach to find permutations to compress classifiers and develop the FPGA-based hardware implementation scheme.

The rest of this paper is organized as follows. Section II reviews the related work. Section III summarizes the problem in previous works and introduces our motivation as well as the BP technique. Section IV defines terms and concepts. Section V proposes a heuristic solution to compress classifiers. Section VI analyzes the hardware implementation of packet classification based on BP. Section VII presents the simulation results. Finally, section VIII concludes the paper. Due to the space limit, more details about the BP are presented in [19].

II. RELATED WORK

Previously-proposed schemes on classifier compression share a common objective that is to find a smaller semantically equivalent classifier for a given classifier by taking advantage of two properties:

[1] Hereafter, we assume that all the classifiers used in the examples have been already expanded to prefixes, and no longer contain ranges.

1) *Action-Oriented.* In packet classification, we can modify a classifier as long as the modification doesn't change the action returned by the classification operation.

2) *First-Matching.* If multiple rules match the given packet, TCAMs natively only return the first matched rule.

Based on these properties, Dong et. al. in [10] proposed four simple heuristic algorithms called *Trimming*, *Expanding*, *Adding* and *Merging*. Liu et. al. proposed an algorithm based on *Firewall Decision Diagram* [11]. Meiner et. al. proposed *Topological Transformation Approach* [12].

Actually, [10] [11] [12] are all field-level schemes, which only focus on each field and fail to explore the compression across different fields. In viewing this, McGeer et. al. proposed a bit-level solution in their work [14] which can yield a higher compression. In this solution, the classifier compression problem is treated as a special logic optimization problem with 104 variables, where each rule in the classifier represents a product of several variables. Therefore, the existing logic optimization techniques can be applied to compress classifiers. Moreover, with the first-matching property of TCAM, the compression can be even better [14].

In this paper, we propose the BP technique, which can achieve significantly higher compression rates compared to McGeer's algorithm. For convenience, in the rest of the paper, all rules in the examples 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. MOTIVATION

A. Rule-Distribution

(a) Dense

WXYZ		YZ \ WX	00	01	11	10
0000	Accept	00	A	D	D	D
0001	Accept	01	A	A	D	D
0011	Accept	11	A	A	D	D
0010	Accept	10	A	A	D	D
0101	Accept					
0111	Accept					
0110	Accept					
****	Deny					

(b) Sparse

YZ \ WX	00	01	11	10
00	D	A	D	A
01	A	D	A	D
11	D	A	D	A
10	A	D	A	D

WXYZ	
0001	Accept
0010	Accept
0100	Accept
0111	Accept
1101	Accept
1110	Accept
1000	Accept
1011	Accept
****	Deny

Figure 1 Typical Rule Distributions (a) Dense (b) Sparse

As we have stated earlier, the recent progress [14] on classifier compression is achieved by logic optimization and the first-matching property. These two methods work well for the *rule distribution* like Figure 1 (a) where *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), but perform badly in "sparse" rule distribution like Figure 1 (b). This observation motivates us to develop the BP technique to convert sparse rule distributions to dense rule distributions before applying the logic optimization and the first-matching property for compression.

B. Block Permutation (BP)

We use a simple example in Figure 2 to demonstrate the main idea of BP. In the example, BP compresses the sparsely-distributed Original Classifier by two simple permutations. In the first permutation, we switch Column "01" and Column "11" in the Original Table. In the second permutation, we switch Row "10" and Row "11" in Table 1. Then by applying logic optimization on Table 2, the original five rules are merged into two rules.

Corresponding to these two permutations, we need to apply two transformations on incoming packets before performing the packet classification operation on TCAMs. In the first transformation, if the WX bits of the packet header are "01" (or "11"), we change them to "11" (or "01"); otherwise, we keep the WX bits unchanged. This transformation and its corresponding permutation is denoted as "01--<>11--" (or "01<>11@WX"). In the second transformation, "--10<>--11" is performed. Obviously, by using the transformed packets to lookup Classifier 2, we can get the same actions as we use the original packets to search the Original Classifier.

Based on this idea, the implementation architecture of BP consists of two modules. Packets should be first processed by a Transformation Module and then fed into a TCAM Module that stores the compressed classifier. For this scheme, we need to consider the following issues:

1) *Processing Speed.* To ensure a high performance, the transformation module should be implemented by hardware.

2) *Overhead.* While BP can reduce the TCAM size, the transformation module does introduce overhead. Fortunately, the overhead is much smaller than the TCAM resource saved (as we will see in Section VII). It is improtant to point out that switching small blocks causes more overhead than switching big blocks. For example, in the second permuation of Figure 2, if we perform "0-10<>0-11", the overhead required for the corresponding transformation will be higher. So, we should switch blocks which are as large as possible when doing the permutation operations.

3) *Programmability.* Because the classifier may require updates from time to time, programmability is another concern. The classifiers usually do not need very frequent updates, normally once every day or several days [18]. We suggest use FPGA to implement the transformation module to achieve the programmability.

IV. TERMS AND CONCEPTS

Before introducing the algorithm of BP, 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. For example, the size of the block "0**1" in Table 2 of Figure 2 is 4. The block size can also be represented by the number of wildcard '*' in the Boolean representation. The more wildcards there are, the larger the block is.

2) *Distance:* The distance of two blocks (or points) in Boolean Space is defined as the number of different non-'*' counterpart bits in their Boolean representations. For example, to calculate the distance between "0*01" and "**00", we first ignore W bit and X bit because these positions contain '*', then find only one different bit, i.e. Z, so the distance is 1.

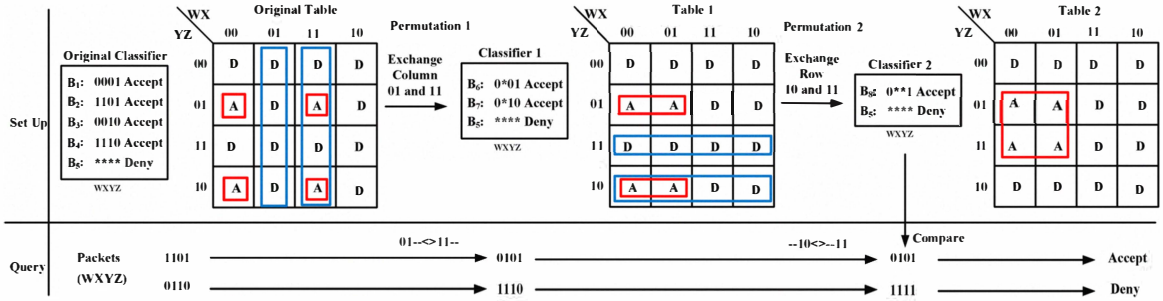

Figure 2 A Simple Example of the BP Technique

3) *Direction*: If the Boolean representations of two blocks have wildcards that all appear in the same positions, 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, i.e. no wildcard in their Boolean representations, are always treated as in the same direction.

4) *Merge and Permutation:* Merge and Permutation are the two basic operations to manipulate blocks in Boolean Space. Only when two blocks meet all the conditions listed in Figure 3, can we perform the corresponding operation on them. Please note that the condition of "Same action" means that all points in the two blocks should be associated with the same action ("deny" or "accept").

Merge	Permutation
(1) Same action	(1) Same action
(2) Same block size	(2) Same block size
(3) Block distance = 1	(3) Block distance >= 2
(4) Same direction	(4) Same direction

Figure 3 Conditions of Merge and Permutation

5) *Target Blocks and Assistant Blocks:* In a permutation, we switch two Assistant Blocks to merge two Target Blocks (the target blocks need to meet the conditions of Permutation in Figure 3). For example, in Table 1 of Figure 2, "0*01" and "0*10" is a pair of target blocks (denoted as "B_6B_7(YZ)"). To merge them, we perform permutation "--10<>--11" over the assistant blocks "**10" and "**11". In this example, we denote the permutation as "--10<>--11" or "10<>11@YZ". Generally, if the assistant blocks are "$* \cdots * a_{i1} \cdots a_{is} b_{k1} \cdots b_{kt}$" and "$* \cdots * a_{i1} \cdots a_{is} \overline{b_{k1}} \cdots \overline{b_{kt}}$", then the permutation is "$a_{i1} \cdots a_{is} b_{k1} \cdots b_{kt} <> a_{i1} \cdots a_{is} \overline{b_{k1}} \cdots \overline{b_{kt}} @ X_{i1} \cdots X_{is} X_{k1} \cdots X_{kt}$", where $X_{i1}, \cdots, X_{is}$ and $X_{k1}, \cdots, X_{kt}$ are the positions of the non-wildcard bits in the Boolean representations of the assistant blocks. Apparently, a pair of assistant blocks specifies a permutation. Normally, to merge two target blocks, there might be multiple pairs of assistant blocks as options. To reduce the overhead, it is wise to choose large assistant blocks.

V. Classifier Compression

In this section, we propose the algorithm of BP in Figure 4 to compress classifiers. There are two phases in the algorithm: the preprocess phase and the permutation phase. In the preprocess phase, we apply logic optimization on the original classifier to group adjacent rule elements together. This is to reduce the rule number involved in the permutation phase and hence reduce the computation complexity. To lower the overhead, in the permutation phase, we recursively search permutations by checking assistant blocks from the size of large to small. If the allowed maximum iteration (indicated by *Nr*) has been reached, or we can't find a valid permutation in the current round of iteration, the program will be terminated.

```
Input:  Classifier C0;
        Maximum number of permutations Nr ;
        Maximum assistant block size Wmax ;
        Minimum assistant block size Wmin ;
Output: Compressed Classifier C1;
Begin
    C1 <- PREPROCESS(C0) ;                                  Phase 1
    For (i=1, 2, ··· , Nr) {                                Phase 2
        For (Wp = Wmax to Wmin) {
            target <- FIND_TARGET(Wp,C1);                   (1)
              If ( target= Φ ) continue ;
            perm <- EVAL_PERM(target,Wp,C1);                (2)
              If ( perm= Φ ) continue;
            C1 <- EXEC_PERM(perm, C1);                      (3)
              break;
        }
        If ( target= Φ or perm= Φ ) return C1;
    }
    return C1;
End
```

Figure 4 Procedure for BP Algorithm

As listed in Figure 4, there are three steps in each round of iteration in the permutation phase, such as FIND_TARGET (Find Targets), EVAL_PERM (Evaluate Permutations) and EXEC_PERM (Execute a Permutation). And there is a parameter *Wp* specifies the expected assistant block size and can be used to reduce the computation complexity by applying the following properties, which disclose the relationship between assistant blocks and target blocks.

Property 1: The size of the assistant block cannot be smaller than the size of the corresponding target block.

For example, in Table 1 of Figure 2, the assistant block "**10" covers the target block "0*10". So the assistant block size is not less than the target block size. Assuming the size of the assistant block is *Wp* wildcards and the size of the target block is *Wt* wildcards, then we have (1):

$$Wp \geq Wt \quad (1)$$

Property 2: The size of the assistant block cannot be larger than the number of bits in a rule minus the distance between the two corresponding target blocks.

Generally, assuming that a rule has *L* bits, the distance of the two target blocks is *D* and the size of each assistant block is *Wp* wildcards, we have (2) (please refer to [19] for proof):

$$Wp \leq (L - D) \quad (2)$$

1) FIND_TARGET

In this step, we don't need to find out all target block pairs, but just those that meet all the permutation conditions listed in Figure 3 and satisfy (1) and (2) with the expected assistant block size *Wp* of the current iteration.

2) EVAL_PERM

In this step, we have two tasks. One is to search all possible permutations for the target block pairs that we have obtained in the previous step. The other is to determine if these permutations are worth to be executed and find out the "best" permutation that can yield the largest gain (gain is the compression minus the overhead).

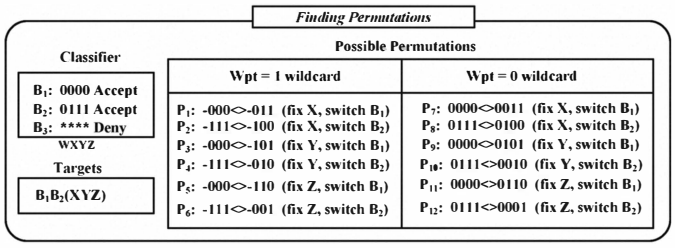

Figure 5 An Example for Finding Permutations for a Pair of Target Blocks

The way to find a permutation for two target blocks is by checking their Boolean representations. Here is an example in Figure 5. Let us consider the two target blocks "B_1B_2(XYZ)". To merge these target blocks, a possible permutation should reduce their distance from 3 to 1. According to (1) and (2), the assistant block size *Wpt* for these target blocks can be *Wpt=1* or *Wpt=0*. Then by fixing one bit to be unchanged and inverting the other bits, we can list all possible permutations (please refer to [19] for more details).

After we find out all possible permutations for a given target, we need to select the "best" one to execute. There are two situations that we need to consider when evaluating permutations. First, one permutation may merge multiple pairs of target blocks. Second, although a permutation can merge target blocks, it might also break some existing blocks, which introduces new blocks. So, the actual compression achieved by a permutation is the number of blocks reduced minus the number of new blocks introduced. If the new blocks are more than the eliminated blocks, then the permutation is considered invalid (please refer to [19] for more details).

3) EXEC_PERM

In this step, we execute the permutation selected in the previous step to merge the target blocks. Consider table 1 in Figure 2. After executing the permutation "--10<>--11", B_7 "0*10" is changed to "0*11" and then merged with B_6 "0*01", resulting a big block B_8 "0**1".

VI. TRANSFORMATION IMPLEMENTATION

As we have explained, if the classifier has been compressed by executing a series of permutations, we need to apply a series of corresponding transformations on the incoming packets. Generally, if we execute the permutation "$a_{i1}\cdots a_{is}b_{k1}\cdots b_{kt} <> a_{i1}\cdots a_{is}\overline{b_{k1}}\cdots\overline{b_{kt}}@X_{i1}\cdots X_{is}X_{k1}\cdots X_{kt}$" (see section IV for the definition) in an *n*-dimension Boolean Space, then the $X_{k1},\cdots,X_{kt}$ bit of the incoming packets need to be transformed. Assuming that the original values of $X_{k1},\cdots,X_{kt}$ are $x_{k1},\cdots,x_{kt}$ respectively, we can calculate their new values after a transformation by the following equations:

$$\begin{cases} x'_{k1} = \overline{x_{k1}}\cdot F + x_{k1}\cdot\overline{F} \\ \vdots \\ x'_{kt} = \overline{x_{kt}}\cdot F + x_{kt}\cdot\overline{F} \end{cases} \quad (3)$$

Where, if $X_{i1}\cdots X_{is} = a_{i1}\cdots a_{is}$ and $X_{k1}\cdots X_{kt} = b_{k1}\cdots b_{kt}$ or $\overline{b_{k1}}\cdots\overline{b_{kt}}$, then $F = 1$; Otherwise, $F = 0$.

Based on (3), we can design circuit on FPGA to implement the transformations. Intuitively, we can use the *Pipeline Structure* to implement a series of transformations. If there are *N* transformations, we can design an *N*-stage pipeline. Or we can design a 1-stage pipeline by merging all transformations together. *N*-stage structure can run at high speed but consumes large hardware resource. 1-stage structure costs less, but the only stage will inevitably become very complicated thus suffer from low speed. Considering the pros and cons of two structures, we propose a solution called *Stage-Grouping* in Figure 6 to achieve the tradeoff between the speed and the cost.

```
Input: A series of transformations P1, P2, ···, PN;
       Targeted clock rate Rt;
Output: Stages S1, S2, ···, SM;
Variables: Current number of groups M;
           Current clock rate R;
Begin
    R<-0;
    Group all transformations into one
    stage and construct a 1-stage pipeline;
    M<-1;
    while(R<Rt) do
        Synthesize the pipeline, get R and
        the worst-performance stage Sw;
        if ( R<Rt ) do
            Split Sw to two sub-stages and
            construct a (M+1)-stage pipeline;
            M<-(M+1);
        end if
    end while
    return Stages;
End
```

Figure 6 Algorithm of Stage-Grouping

Figure 6 shows the proposed algorithm of stage-grouping. The stage-grouping starts from a 1-stage pipeline. In other words, we first merge all transformations into a single stage and synthesize it to estimate the clock rate performance. If the estimated clock rate is faster than the targeted clock rate, the obtained pipeline will be accepted and the stage-grouping will ends. Otherwise, we will split the stage as evenly as possible into two sub-stages, then construct and synthesize a new 2-stage pipeline. If there are multiple stages, we will split the worst-performance one. So on and so forth, we can finally get a well-balanced structure. The way to evenly split a stage is by checking the assistant block sizes of all permutations encapsulated. For example, if a stage incorporates k $(k > 1)$ consecutive permutations whose assistant block sizes are $m_1,\cdots,m_k$ respectively, we need to find the t $(1 \le t < k)$, such that $\sum_{i=1}^{t} m_i$ and $\sum_{i=(t+1)}^{k} m_i$ are as close as possible.

VII. EXPERIMENTS

Our experiments are based on seven artificial classifiers generated by ClassBench [16] and one real-life firewall classi-

Table 2 Classifier Statistics and Results from BP Compression Experiments

Source	Classifier	Statistics			Classifier Compression							FPGA Implementation							
		Rules	Prefixes	Rate	Preprocess Comp.	Comp. Rate	BP Comp.	Comp. Rate	Total Comp.	Comp. Rate	# of Perms	TCAM saved Entries	TCAM saved Gate Count	FPGA consumed CFs	FPGA consumed Registers	FPGA consumed Gate Coun	Ratio	Pipeline Stages	Pipeline Clock Rate
Class-Bench	acl-1	187	357	1.91	50	14.01%	139	38.94%	189	52.94%	79	139	72280	2346	1560	16398	22.69%	15	114.53
	acl-2	217	271	1.25	1	0.37%	154	56.83%	155	57.20%	134	154	80080	4389	1872	24399	30.47%	18	101.36
	acl-3	221	312	1.41	3	0.96%	66	21.15%	69	22.12%	57	66	34320	1335	936	9621	28.03%	9	118.12
	fw-1	60	115	1.92	69	60.00%	12	10.43%	81	70.43%	7	12	6240	69	104	831	13.32%	1	395.57
	fw-2	132	277	2.10	173	62.45%	23	8.30%	196	70.76%	13	23	11960	50	208	1398	11.69%	2	216.59
	ipc-1	202	584	2.89	14	2.40%	237	40.58%	251	42.98%	101	237	123240	2813	1768	19047	15.46%	17	114.29
	ipc-2	207	538	2.60	0	0.00%	326	60.59%	326	60.59%	121	326	169520	3469	1768	21015	12.40%	17	101.36
Real-life	firewall-1	660	807	1.22	295	36.56%	148	18.34%	443	54.89%	59	148	76960	967	832	7893	10.26%	8	109.19
	Avg.	235.75	#####	1.91	75.63	22.09%	138.13	31.90%	213.75	53.99%	71.38	138.13	71825	1929.8	1131	12575	18.04%	10.9	158.88

fier obtained from ISP. The sizes of eight classifiers vary from 60 rules to 660 rules. The average prefix expansion ratio is 1.91. For classifier compression process, in the preprocess phase, we use the *Espresso* algorithm [15] to do logic optimization. In BP phase, we set $Nr = 150, Wmax = 102,$ $Wmin = 54$ and run the program on a Linux workstation driven by *Intel 2.0GHz E5335* CPUs.

The results of our experiments are presented in Table 2. On average, the BP technique can reduce prefixes by 53.99%, among which the preprocess phase contributes 22.09% and the permutation phase contributes 31.90%. For the IPC classifiers, while the permutation phase can save 50.59% prefixes on average, the preprocess phase can barely give any compression. This is because the rule distributions of IPC classifiers are very "sparse", so logic optimization in preprocess phase works poorly. This is what motivated our research on the BP technique. For the very "dense" FW classifiers, permutation phase can still contribute 9.37% compression. For those rule distributions between "dense" and "sparse", like ACL classifiers and the real-life classifier, the permutation phase can give significant compression.

In FPGA implementation process, we set the targeted throughput to 100M packets per second and implemented the transformations on *Altera Cyclone III* FPGA by using the *Quartus II* synthesis tool. To estimate the hardware resource saved by using BP technique (TCAM entries reduced minus FPGA resource consumed), we used the concept of "Equivalent Gate Count". From the TCAM chip ICFWTNM1 [17], 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 (4):

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

The Altera FPGA resource consumption is reported in Combinational Functions (CFs) and Registers. We calculate the FPGA gate count using (5):

$$FPGA\ Gate\ Count = \#\ of\ CFs \times 3 + \#\ of\ Registers \times 6 \quad (5)$$

Experiments show that the average gate count of FPGA consumption is only 18.04% of that of TCAM saved.

In the experiments, the average run-time of compression processes is 15.007 minutes. A classifier with more prefixes and a higher compression ratio requires a longer run-time. For FPGA implementation, the average run time for the synthesis is 20.75 minutes. More pipeline stages require more run-time.

VIII. 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 under the circumstances that direct logic optimization and the first-matching property perform poorly. The improvement is achieved by using a series of permutations to change the rule distribution in Boolean Space. The proposed BP is a new technique for logic optimization. It is not limited to 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] R. Draves, C. King, S. Venkatachary, and B. Zill, "Constructing optimal IP routing tables," in *Proceedings of IEEE INFOCOM*, 1999.

[9] S. Suri, T. Sandholm, and P, "Warkhede. Compressing two-dimensional routing tables," *Algorithmica*, 35:287–300, 2003.

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

[11] A. X. Liu, E. Torng, and C. Meiners, "Firewall compressor: An algorithm for minimizing firewall policies," in *INFOCOM*, 2008.

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

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

[14] R. McGeer and P. Yalagandula, "Minimizing Classifiers for TCAM Implementation," in *IEEE INFOCOM*, 2009.

[15] http://en.wikipedia.org/wiki/Espresso_heuristic_logic_minimizer

[16] http://www.arl.wustl.edu/~det3/ClassBench/index.htm

[17] http://www.ece.uwaterloo.ca/~cdr/www/chip.html

[18] http://www.snort.org

[19] http://eeweb.poly.edu/chao/publications/TechReports.html