A Fast Regular Expression Matching Engine for NIDS Applying Prediction Scheme

Lei Jiang
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, P.R. China
University of Chinese Academy of Sciences
Beijing, P.R. China
Email: jianglei@ict.ac.cn

Qiong Dai, Qiu Tang, Jianlong Tan
Institute of Information Engineering
Chinese Academy of Sciences
Beijing, P.R. China
Email: daiqiong@iie.ac.cn
tangqiu@iie.ac.cn
tanjianlong@iie.ac.cn

Binxing Fang
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, P.R. China
Email: bxfang@ict.ac.cn

***Abstract*—Regular expression matching is considered important as it lies at the heart of many networking applications using deep packet inspection (DPI) techniques. For example, modern networking intrusion detection systems (NIDSs) typically accomplish regular expression matching using deterministic finite automata (DFA) algorithm. However, DFA suffers from the high memory consumption for the state blowup problem. Many algorithms have been proposed to compress the DFA memory storage space, meanwhile, they usually pay the price of low matching speed and high memory bandwidth. In this paper, we first propose an effective DFA compression algorithm by exploiting the similarity between DFA states. Then, we apply a next-state prediction strategy and present a fast DFA matching engine. Carefully designing the DFA matching circuit, we keep the prediction success rate by more than 99.5%, thus get a comparable matching speed with original DFA algorithm. On the side of memory consumption, experimental results show that with typical NIDS rule sets, our algorithm compressed the original DFA by more than 99%. Mapping this algorithm on Xilinx Virtex-7 FPGA chip, we get a throughput of more than 200Gbps.**

I. Introduction

Regular expression matching lies at the heart of deep packet inspection (DPI)[1] applications, especially for the Networking intrusion detection systems (NIDSs). Modern NIDS, such as Snort [2] and L7-filter [3], use regular expression rules to detect networking attacks. Compared with the simple string rules, regular expression rules have higher expressive power and are able to describe a wider variety of payload signatures [4]. State-of-the-art NIDS uses DFA algorithm to perform regular expression matching for its line rate matching speed. But as the rule sets become complex and large, DFAs suffer from the state blowup problem, especially for the patterns with constrained and unconstrained repetitions of wildcards and large character sets [5]. According to [6], the L7-filter's rule set, containing 109 regular expression rules, consumes more than 16GB memory space when compiled to a composite DFA. Compression mechanism is an effective way to reduce memory consumption of DFA. Many compression algorithms have been proposed, such as D^2FA [7], δFA [8][9] and A-DFA [10]. These algorithms use the redundancy of DFA transition table to generate a new compressed DFA structure. Meanwhile, the compression of DFA implies that multiple states may be traversed when processing a single input character. So the compression algorithms usually pay a price of worse memory bandwidth and lower matching speed.

In this paper, we continue focusing on the DFA compression mechanism and develop a new DFA compression algorithm called J-DFA. We apply clustering algorithm to classify all DFA states to different groups. In each group, we extract a common state, and the transitions in this group different from the common state are stored in a sparse matrix. Then, we encoded the common state by run-length encoding. By using these methods in combination, the compression ratio of J-DFA reaches 99%.

The key issue of mapping DFA compression algorithm into FPGA is how to access the compressed DFA structure. After compressing, the DFA transition table becomes irregular because a lot of zero-elements are eliminated. Previous works focus on the compressing technologies and place little emphasis on how to access the irregular compressed transition table efficiently. Only in [11], bitmap is mentioned to store the compressed DFA structure. However, bitmap method consumes at least 3 clock cycles to accomplish one lookup, thus greatly decreasing the matching speed. So, we present a novel architecture to resolve the conflict between memory usage and matching speed. We design a state prediction method to accelerate regular expression matching based on J-DFA algorithm. We observe that in the real matching process of J-DFA, it has a great chance that the "next state" lies in the same "clustering group" of the "current state". So we can predict the "next state" according to the "clustering center" of the "current state". Inspired by the locality principle of programs behaving in memory and CPU cache [12][13], we design a next-state prediction unit [14][15] and add it to our regular expression matching engine on Xilinx Virtex-7 FPGA chip. Experiment results show that the prediction success rate is more than 99.5%, thus achieving a comparable matching speed with original DFA algorithm.

In summary, the main contributions of this paper are:

(i) We develop a new DFA compression algorithm called J-DFA by clustering algorithm and encoding scheme. Moreover, we measured the compression ratio of J-DFA. Measurement results show that the compression ratio reaches about 99%.

(ii) We develop a state prediction method for J-DFA and measured it using real-life NIDS regular expression

rulesets and datasets. Measurement results show that the prediction success rate of J-DFA reaches more than 99.5%.

(iii) Based on J-DFA, we design a regular expression matching engine with state prediction unit. This engine performs very well in both memory usage and memory bandwidth. Using parallel lookups on the newest Xilinx Virtex-7 chip, the throughput reaches from 230Gbps to 430Gbps for real-life rulesets.

II. Related Work

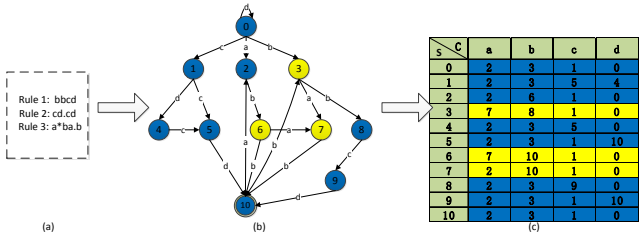
Kumar et al.[7] observed that two states (S1, S2) have many similar next state transitions (T) for an input characters subset. According to this observation they proposed a new algorithm called D^{2}FA to compress the transition table. D^{2}FA eliminates S1's transitions (T) by introducing a default transition from S1 to S2. The experimental results show that a D^{2}FA reduces transitions by more than 95% compared to original DFA. However, D^{2}FA's transition mechanism is possible to look up memory multiple times per input character, leading to a higher memory bandwidth.

To avoid the high memory bandwidth requirement of D^{2}FA, Ficara et al.[8] present a new representation for deterministic finite automata, called Delta Finite Automata (δFA). They record the transition set of current state into a local memory, and only store the differences between current state and next hop state. In this way, δFA achieves very good compression effect. In addition, this algorithm requires only a state transition per character (keeping the characteristic of standard DFAs), thus allows a fast string matching speed. But δFA does not optimize the DFA structure, resulting in a relatively weak compression ratio compared with other algorithms.

Becci et al. improved the idea of D^{2}FA, and developed a DFA compression algorithm called A-DFA in 2013 [10]. By introducing the notion of "state depth" to quantify a state's distance from the initial state, A-DFA constructs nearly optimal default paths. Compared with D^{2}FA, A-DFA results in at most 2N state traversals when processing an input string of length N, and yields a tenfold improvement in compression ratio. A-DFA is a nearly optimal transition-merging algorithm in both memory usage and bandwidth.

All the algorithms mentioned above used the redundancy of DFA's transition table. Besides these, many other algorithms were presented in recent years: T. Liu et al .[6] divided all the transitions into three groups and stored them into three different matrixes, then compress these matrixes. Y. Liu et al .[16] decomposed the DFA transition table into a column, a row vector and a sparse matrix to reduce the storage space as much as possible. QI Y et al.[11] proposed a two-dimensional compression algorithm, utilizing the intra-state redundancy and inter-state redundancy of DFA to reduce the memory consumption.

In our opinion, the essential of DFA compression algorithms is how to balance the memory consumption, the memory bandwidth and the matching speed. Compared with algorithms mentioned above, J-DFA combined state encoding with the transition redundancy elimination technique to get an ideal compression ratio. Besides, using the state prediction scheme, J-DFA significantly improved the matching speed of regular expression matching engine. In the following sections, we will show the technique detail of J-DFA.



S \ C	a	b	c	d
0	2	3	1	0
1	2	3	5	4
2	2	6	1	0
3	7	8	1	0
4	2	3	5	0
5	2	3	1	10
6	7	10	1	0
7	2	10	1	0
8	2	3	9	0
9	2	3	1	10
10	2	3	1	0

Fig. 1. A complex case of $(bbcd)$, $(cd.cd)$ and $(a*ba.b)$. DFA states are divided to two groups. (a) is the regex rules; (b) is the corresponding DFA; (c) is the transition table.

III. J-DFA Algorithm

J-DFA algorithm is based on this observation: the states of DFA from real-life regular expression rule-sets have an obvious characteristic of clustering. More specifically, almost every state has multiple similar states, with very little different transitions between them, so the transition table can be divided into one or more groups by putting similar states together. To better illustrate this phenomenon, we use one DFA generated from regex rules of $(bbcd)$, $(cd.cd)$ and $(a*ba.b)$. This DFA is shown in Fig.1. For simplicity, the default transitions are omitted in Fig.1(b). Fig.1(c) is the corresponding transition table. It is very obvious that there are a lot of similar states in Fig.1(c). And we group these states into two groups, with each group colored as yellow and blue.

From the phenomenon in Fig.1(c), we develop a DFA compression algorithm called J-DFA. J-DFA algorithm has two steps: first, to decompose a DFA using clustering algorithm, and second, to further compress the storage space using runlength encoding scheme. In the first step, we use the classical data clustering algorithm to group the similar states, and the DFA states is divided into multiple groups. Meanwhile, the center state of each clustering could be gained. Then, we extract the special transitions by subtracting each state with the clustering center state. So the original DFA transition table is converted into multiple clustering center states (common states) and small amount of special transitions. Now the original DFA is decomposed to three parts: (i) Index Table; (ii) Clustering Groups; (iii) Sparse Matrix, as in Fig.2. In Fig.2, we illustrate the decomposition still using the example in Fig.1. We can see that the DFA states are classified to four groups, every group with one common states. We put these four common states together, calling it "Clustering Groups". And the transitions different with common states are left in the table, and we call this table "Sparse Matrix". Finally, we build an "Index Table" to indicate the relationship between original states and the common states.

In the second step, we further compress the storage space by encoding the common states. We perform the runlength encoding to the Index Table and the Clustering Groups of Fig.2. Assuming a common state is $aaaaabbbccccddddddd$, then, after the run length encoding, the common state becomes $a\{5\}b\{3\}c\{4\}d\{6\}$, which means a repeating 5 times, the b repeating 3 times, and so on. Experiment results show that the

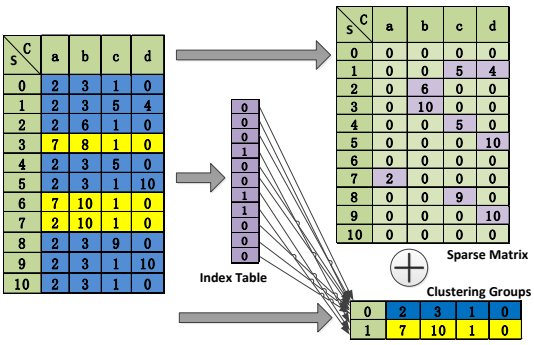


Fig. 2. A DFA is decomposed to three parts: (i) Index Table; (ii) Clustering Groups; (iii) Sparse Matrix.

runlength encoding can improve the compression ratio from about 3% to less than 1%.

Algorithm 1. : Construction Algorithm of J-DFA

Input: $TransitionTable, N, C, K$
Output: $ClusteringGroups, SparseMatrix, IndexTable$
1: $IndexTable \leftarrow KMeans(TransitionTable)$
2: **for** $i = 1$ to K **do**
3: $ClusteringGroups[i] \leftarrow getCenter(TransitionTable, i)$
4: **end for**
5: **for** $s = 1$ to N **do**
6: $k \leftarrow IndexTable[s]$
7: **for** $c = 1$ to C **do**
8: $SparseMatrix[s][c] \leftarrow TransitionTable[s][c] - ClusteringGroups[k]$
9: **end for**
10: **end for**
11: **for** $i = 1$ to K **do**
12: $ClusteringGroups[i] \leftarrow runlengthEncoding(ClusteringGroups[i])$
13: **end for**
14: $IndexTable \leftarrow runlengthEncoding(IndexTable)$

Detailed construction algorithm of J-DFA is listed in Algorithm 1. N is the number of DFA states. C is the size of character set. K is the number of groups that DFA states are divided. In Algorithm 1, because we adopt the K-means algorithm, it is hard to determine the initial K. So we manually try different K for a rule set, and find the best result. In the 1st line, we divide all the DFA states into K groups by K-means clustering algorithm, and $IndexTable$ is worked out to indicate which group every DFA state s belongs to. From line 2 to line 4, we calculate the common states for each group by $getCenter$ function. Details of this function is shown in Algorithm 2. From line 5 to line 10, we visit every state of DFA, and store the different transitions to $SparseMatrix$. From line 11 to line 14, we further compress the $ClusteringGroups$ and $IndexTable$ using runlength encoding. The time complexity of Algorithm 1 is $O(K * N * C)$. During the construction, we need two $N * C$ matrix to deposit the transitions. So the space complexity of Algorithm 1 is $O(N * C)$.

Algorithm 2 presents how to calculate a common state from a group. The principle of calculating common state is to guarantee the $SparseMatrix$ as sparse as possible. Thus, for every c in alphabet, we must select the most frequent transition as a common state transition.

Algorithm 2. : Pseudocode for $getCenter$ function

Input: $Group, C$
Output: $CommonState$
1: $L \leftarrow getArrayLength(Group)$
2: **for** $c = 1$ to C **do**
3: $Array.clear()$
4: **for** $l = 1$ to L **do**
5: $Array.insert(Group[l][c])$
6: **end for**
7: $CommonState[c] \leftarrow findMostFrequentElement(Array)$
8: **end for**

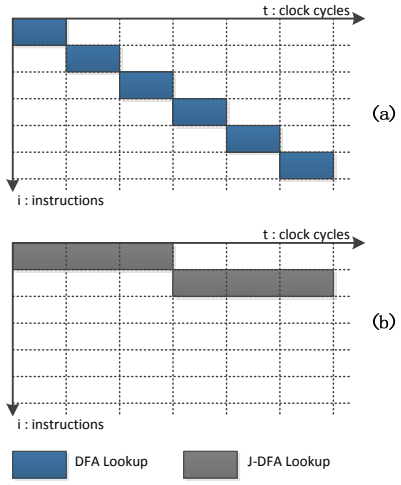


Fig. 3. Pipeline diagram of DFA and J-DFA.

IV. State Prediction Technique

In this section, we talk about the state prediction technique used in our regular expression matching engine. This technique relies on the run-time property of J-DFA. In the following paragraphs, we first elaborate why DFA compression algorithms decrease the matching speed by analyzing the lookup procedure of J-DFA. Then we present the details of state prediction technique, and verify its effectiveness by experiment results.

A. Lookup Procedure of J-DFA

Usually, DFA completes one lookup in one clock cycle on FPGA. However, after compressing, the lookup procedure becomes complex. When conducting a state lookup in J-DFA, we have to perform the following steps. First, we search the "Index Table" in Fig.2 for the common state, and get the "next state" from the common state. Then, we look up the "Sparse Matrix" to determine whether the corresponding transition is a "non-zero element". If it is a "non-zero element", this value is assigned to "next state". Of all these steps, the sparse matrix lookup is the most time-consuming operation, consuming at least 3 clock cycles. That means, we must consume at least 3 clock cycles to finish one J-DFA lookup.

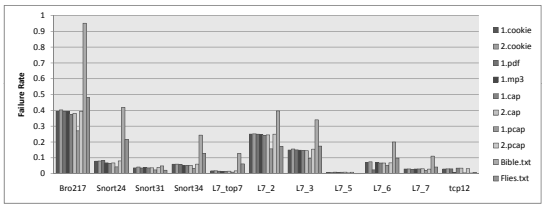


Fig. 4. Prediction failure rate using normal DFA state.

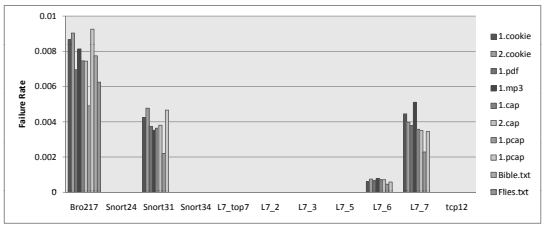


Fig. 5. Prediction failure rate using common state

Compared with the lookup procedure of normal DFA, the lookup procedure of J-DFA is complex and more time-consuming. When implemented on FPGA, it costs more than 3 clock cycles to complete a lookup for J-DFA. Fig.3 shows the pipeline diagram of DFA and J-DFA. In all the six clock cycles, the DFA algorithm conducts the lookup operation for 6 times, while J-DFA algorithm only conducts the lookup operation for twice. Because the lookup operation is a self-feedback procedure: the input signal "current state" of DFA is determined by the output signal "next state", the pipelining technique is invalid for DFA lookup. So the lookup operation of J-DFA in Fig.3(b) has to pause for 3 clock cycles waiting for the previous lookup operation to finish. Obviously, the matching speed of J-DFA is 3 times slower than that of DFA.

B. Details of State Prediction Technique

To reduce the clock delay brought by DFA transition table compression, we proposed a state prediction approach in our regular expression matching design. This approach is based on two observations: first, most adjacent states share a large part of the same transitions. Here, if one state can traverse to another state, we call the two states are "adjacent". Second, in a real matching process of J-DFA, if "current state" is in one "clustering group", it has a great chance that "next state" is still in the same "clustering group". This means at any moment if one "common state" is used, it is very likely be used in the next clock cycle.

The first observation has been widely used and proved in many previous DFA compression algorithms to compress the DFA transition table, such as D^{2}FA[9] and δFA[10]. The second observation is the run-time property of J-DFA, and we call this property the "locality" of J-DFA. Using these two observations, we can predict the "next state" with a high success rate.

1) Measurement of State Prediction: To validate our observations, we measure the success rate of J-DFA applying prediction scheme. The measurement is based on several real-life rulesets from Bro, Snort and L7-filter. We use the opensource $regex{-}tool$, which is provided by Michela Becchi in [17], to generate the DFAs. L7-filter's ruleset is hard to generate a whole DFA for the severe state blowup problem, so we divide the L7-filter ruleset to multiple groups using the Yu's grouping method[5]. Also the snort rule set is divided into multiple groups. Details of rule sets are shown in Table I. The test datasets are taken from six different sources. These datasets include plain text, music, format document, and network traffic traces. More details of test data are shown in Table II. The cookies are collected from the Google Chrome application. The cap_1 and cap_2 datasets are traces from the campus network. The pcap_1 and pcap_2 datasets are traces captured from the family network. For NIDS, the traffic traces are the most universal and representative. However, considering the extreme cases in network, we also adopt the application data which may be inspected by NIDS in our measurement.

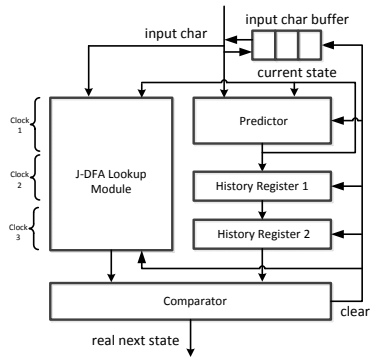


Fig. 6. J-DFA lookup procedure applying state prediction.

2) Experiment Design: To exploiting the "locality" of J-DFA, we designed a simple cache scheme. We first keep recently used states in a set of small but fast caches. When conducting a new state lookup operation, instead of looking up the whole transition table stored in memory, we get the next state from cache. For normal DFA, we store the "current state" in the cache. While for J-DFA, we store the "common states" in the cache. Next, we conduct two experiments to predict the next state, separately for normal DFA and J-DFA.

3) Measurement for Normal DFA: Detailed process of this experiment is as follows: during the progress of state traverse, the "previous state" is named S_{cache} and the "current state" is named $S_{current}$. S_{cache} is recorded in cache to predict the next state transition. When the input char c arrives, if the next transition $S_{cache}(c)$ not equal to $S_{current}(c)$, the prediction fails. And the failure counter is incremented by 1. In every clock cycle, $S_{current}$ is loaded into cache to replace the outdated S_{cache}. In this way, we get the failure rate of normal DFA prediction for various rulesets and datasets in Fig.4. We can make conclusion: to some extent, the normal DFA state transition follows the locality principle. In most cases, the failure rate of prediction is lower than 40%. However, for plain text datasets, the experiment results is so bad. Especially, for "Bible.txt" in "Bro217", the failure rate reaches as high as 95%. Altogether, the prediction scheme on normal DFA is not very efficient, and can not meet the complex business requirements.

4) Measurement of J-DFA: Similar with normal DFA, details of J-DFA experiment is as follows: during the progress of state traverse, the "previous state" is named S_{cache} and the current "common state" is named S_{common}. S_{cache} is recorded

TABLE I. CHARACTERISTIC OF RULESETS

rulesets	rules number	states number
bro217	217	6533
snort24	24	8335
snort31	40	4864
snort34	34	9754
l7_top7	7	12910
l7_2	7	1888
l7_3	12	2293
l7_5	6	2984
l7_6	5	4887
l7_7	5	4028

TABLE II. CHARACTERISTIC OF TEST DATA

data type	size(KB)	state traverse times
cookie_1	6,670	6,765,306
cookie_2	14,726	15,079,191
pdf	1,551	1,587,858
mp3	3,786	3,876,227
cap_1	20,480	20,971,520
cap_2	4,858	4,974,480
pcap_1	8,896	9,109,435
pcap_2	28,667	29,354,030
Bible.txt	4,128	4,227,016
Flies.txt	1,104	1,130,314

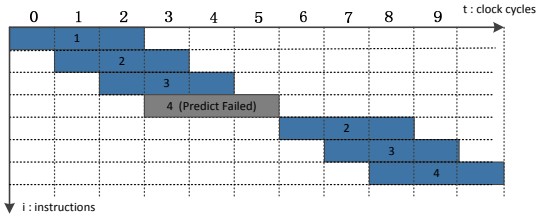


Fig. 7. Pipeline diagram of J-DFA with predictor.

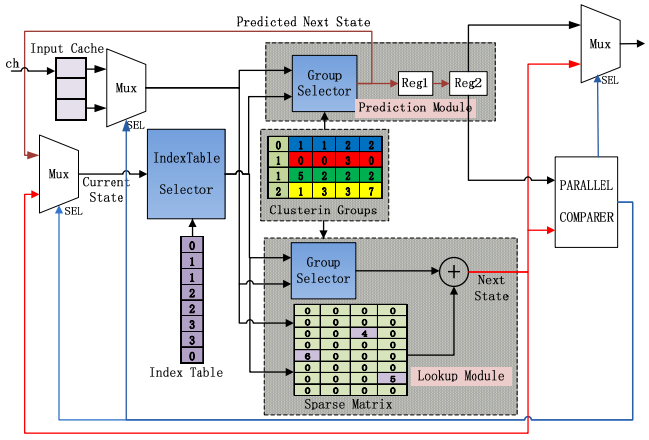


Fig. 8. Framework of regular expression engine

in cache to predict the next state transition. When the input char c arrives, if the next transition $S_{cache}(c)$ not equal to $S_{current}(c)$, the prediction fails. And the failure counter is incremented by 1. Instead of the "previous state", the "common state" is reserved in cache. In every clock cycle, S_{common} is loaded into cache to replace the outdated S_{cache}.

J-DFA improves the prediction success rate using the common state in "Clustering Groups" to predict the next state. Prediction failure rate of J-DFA is shown in Fig.5. Expect Bro217, the failure rate of J-DFA is lower than 0.5%. For some cases, the failure rate is 0%, which means every prediction succeeds. Obviously, this result is better than that of normal DFA. And the prediction scheme on J-DFA is rather efficient and steady. So in the following section, we implement a regular expression matching engine based on J-DFA and state prediction scheme.

V. HARDWARE IMPLEMENTATION

A. Mapping of State Prediction Scheme

From previous discussion, the sparse matrix lookup operation can be pipelined into 3 steps. The lookup procedure of J-DFA applying state prediction is shown in Fig.6. Every time one character comes, the "Predictor" generate a "Predicted Next State" as the next state. This operation only costs one clock cycle. At the same time, the "real" next state is calculated in the "J-DFA Lookup Module". The "J-DFA Lookup Module" is designed as a 3-stage pipeline. So the "Predictor" must record the results of recent 3 clock cycles in "History registers". If the "Predicted Next State" dose not equal to the "real next state", the prediction of 3 clock cycles ago fails. In this case, all the results recorded in the "History registers" is cleared and the states of the whole circuit is restored to that of 3 clock cycles ago. So we must buffer the recent 3 input character in the "Input Char Buffer".

Fig.7 is the pipeline diagram of J-DFA with predictor. Compared with Fig.3(b), the 2nd lookup operation starts at the 2nd clock cycle without waiting for the finish of the 1st lookup operation. This is because we use the predicted state to perform the next matrix lookup operation applying the state prediction technique. If the prediction fails, as at the 4th clock cycle in Fig.7, the whole pipeline must restore to 3 clock cycles ago. After clearing the "history registers", the pipeline restarts from the 2nd lookup operation at the 7th clock cycle. From this example, we can see that when the prediction fails, the pipeline must pay a punishment of extra 3 clock cycles.

B. J-DFA Regex Matching Engine

Detailed architecture of J-DFA regular expression matching engine is shown in Fig.8. As is said in section IV, original DFA is divided into three parts: index table, clustering groups and sparse matrix (Fig.1). The index table is mapped into the "Index Table" module of Fig.8. The clustering groups is mapped into the "Clustering Groups" module. The sparse matrix is mapped into the "Sparse Matrix" module. After using the prediction scheme to decrease the clock cost, the average clock cycles that the pipeline costs can be calculated by this equation: $ACC = NC * p + PC * (1 - p)$. "$ACC$" means "average clock cycles" of the pipeline. "NC" means clock cycles consumed in normal case. In our design, the value of "NC" is 1. "p" means the prediction failure rate. "PC" means "punishment clock cycles" of the pipeline if the prediction fails. In our design, the value of "PC" is 3.

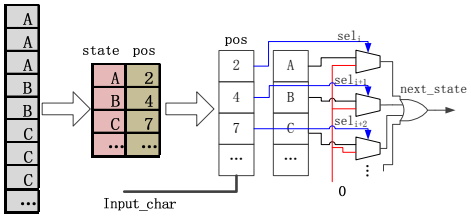


Fig. 9. Diagram of Runlength Decoder Module

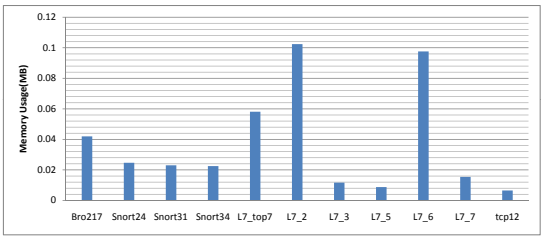


Fig. 10. Memory Usage of J-DFA for Bro, Snort and L7-filter rulesets

In our design, index table and clustering groups are runlength encoded though not shown in Fig.8. And we designed a runlength decoder unit for these two modules (blue boxes in Fig.8). The decoder is implemented using multi-path comparators, as is shown in Fig.9. For example, if now the "input char" is 3, then this value is compared with the pos array simultaneously. If $pos[i] \geq 3$ and $pos[i-1] < 3$, i is the right value decoded. In this example, i is 1, and the state decoded is "B". In this way, the index table and clustering groups state can be decoded within one clock cycle.

C. Performance Evaluation

We target the regular expression matching engine into a Xilinx Virtex-7 FPGA chip(XC7VX1140T: 1,139,200 logic cells (LCs), 17,700 Kb Distributed RAM, total 67,680 Kb BRAM). Rulesets and datasets have been presented in Section III and Section IV. We evaluate our regular expression matching engine from these aspects: memory usage, memory bandwidth and throughput.

1) Memory Usage: In this section, we compared memory usage of J-DFA with three other DFA compression algorithms: δFA[8], D^{2}FA[7] and CRD(colomn-row decomposition)[16] algorithm. All these algorithms aim at eliminating the redundancies of DFA transition table, which is similar to J-DFA.

The compression ratio of these algorithms is presented in Fig.11 and Table III. For most of rulesets, the compression ratio is lower than 1%. This result yields a tenfold improvement than previous work like D^{2}FA and δFA. This is because J-DFA not only reduces the redundancy between DFA states, but also eliminates the redundancy within common states by runlength encoding scheme. The total memory usage includes "Index Table", "Clustering Groups" and "Sparse Matrix". We suppose: one state costs 14bits; one character costs 8 bits; sparse matrix position costs 14bits. Fig.10 shows how much memory space each ruleset costs. From Fig.10, the memory usage is less than 0.1MB, and there are about 8.26MB block memory on FPGA chip. Conservatively estimating, we can run more than 80 sets of L7_2 ruleset in parallel. Because there are 7 rules in L7_2 ruleset, we can run about 80*7=560 L7-filter rules overall. By similar calculations, we can run more than 8,000 snort regex rules, or 40,000 bro regex rules on Virtex-7 FPGA chip.

2) Memory Bandwidth: The memory bandwidth is determined by the efficiency of memory access and the number of extra bits used for locating the compressed DFA. For our design, if the prediction fails, the punishment is 3 clock cycles, the memory access times can be calculated as $(1 - failure\_rate) * 1 + failure\_rate * 3$. In Fig.12, we compared the the average state travel times per input character

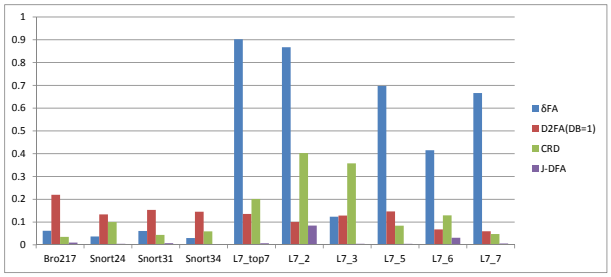


Fig. 11. Compression Ratio of δFA, D^{2}FA, FEACAN and J-DFA

with normal DFA, δFA, D^{2}FA, and FEACAN[11]. FEACAN uses both intra-state and inter-state compression techniques to compress the DFA table, and uses bitmap to store the compressed transition table. From Fig.12, we can see that J-DFA nearly not adding the memory access times compared with normal DFA. Total bits for one transition of J-DFA includes these aspects: input char and state for sparse matrix, common state and predictor and auxiliary bits for looking up sparse matrix. If the auxiliary bits is 32 bits (16+16), input char is 8 bits, state is 16 bits, the bandwidth of J-DFA for one transition is about (8+16)*3+32=108. This result is acceptable for State-of-the-art FPGA chips and blockrams.

3) Throughput: We write the regular expression matching engine in *Verilog* language and implement the prototype on Virtex-7 FPGA chip. We simulate our design using Modelsim6.5 simulator, and synthesize with ISE14.2 synplify tool chain. The max frequency for different rulesets is more than 200MHz, and we take 150MHz to moderately evaluate our throughput. We achieve scalable performance by using parallel regular expression engines. Because J-DFA is a memory-based design, the number of parallel engines is limited by the overall size of blockRAMs available on FPGA. Table IV shows the

rule sets	δFA	D^{2}FA(DB=1)	CRD	J-DFA
bro217	0.062	0.220	0.035	**0.0100**
snort24	0.037	0.134	0.100	**0.0046**
snort31	0.061	0.153	0.044	**0.0074**
snort34	0.030	0.145	0.059	**0.0036**
l7_top7	0.902	0.136	0.201	**0.0070**
l7_2	0.867	0.101	0.403	**0.0846**
l7_3	0.124	0.129	0.358	**0.0048**
l7_5	0.697	0.147	0.084	**0.0046**
l7_6	0.415	0.068	0.129	**0.0312**
l7_7	0.666	0.060	0.048	**0.0060**

TABLE III. COMPRESSION RATIO OF δFA, D^{2}FA, CRD AND J-DFA

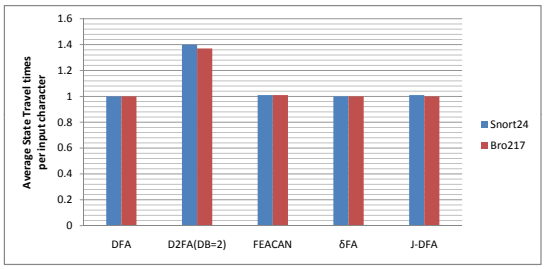


Fig. 12. Memory access comparasion

Rule sets	Memory Usage for Single Engine(KB)	# of engines	f_{max} (MHz)	Throughput (Gbps)
bro217	11.4	196	150	230
snort24	8.9	225	150	264
snort31	6.86	359	150	421
snort34	9.11	367	150	430

TABLE IV. THROUGHPUT OF REGULAR EXPRESSION MATCHING ENGINE

maximum number of engines can be implemented on one Virtex-7 chip with 8640KB blockRAMs. We can see J-DFA engine achieves a throughput of 230∼430Gbps.

VI. CONCLUSION

The tradeoff between memory compression and matching speed is the essential of DFA compression algorithm. Previous works focus on the DFA compression algorithms, but place little emphasis on how to access the irregular compressed transition table efficiently. In this paper, we succeed balancing this conflict using state prediction technique. Our DFA compression algorithm(J-DFA) has very good run-time property suitable for state prediction. Measurement results show that the prediction success rate of J-DFA is higher than 99.5%. Based on J-DFA and state prediction technique, we propose a novel regular expression matching engine. On Xilinx Virtex-7 chip, this engine gets a throughput of 230∼430Gbps based on snort and bro regex rules.

ACKNOWLEDGMENT

This work is supported by National High Technology Research and Development Program of China (863 Program, no. 2012AA012502); the Special Pilot Research of the Chinese Academy of Sciences under grant XDA06030200; and National Natural Science Foundation of China (61303260).

REFERENCES

[1] I. Dubrawsky, "Firewall evolution-deep packet inspection," *Security Focus*, vol. 29, 2003.

[2] M. Roesch *et al.*, "Snort-lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX conference on System administration*. Seattle, Washington, 1999, pp. 229–238.

[3] J. Levandoski, E. Sommer, M. Strait *et al.*, "Application layer packet classifier for linux," 2008.

[4] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 262–271.

[5] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*. IEEE, 2006, pp. 93–102.

[6] T. Liu, Y. Yang, Y. Liu, Y. Sun, and L. Guo, "An efficient regular expressions compression algorithm from a new perspective," in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 2129–2137.

[7] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 339–350, 2006.

[8] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. Di Pietro, "An improved dfa for fast regular expression matching," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 5, pp. 29–40, 2008.

[9] D. Ficara, A. Di Pietro, S. Giordano, G. Procissi, F. Vitucci, and G. Antichi, "Differential encoding of dfas for fast regular expression matching," *IEEE/ACM Transactions on Networking (TON)*, vol. 19, no. 3, pp. 683–694, 2011.

[10] M. Becchi and P. Crowley, "A-dfa: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 4:1–4:26, Apr. 2013.

[11] Y. Qi, K. Wang, J. Fong, Y. Xue, J. Li, W. Jiang, and V. Prasanna, "Feacan: Front-end acceleration for content-aware network processing," in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 2114–2122.

[12] J. R. Spirn and P. J. Denning, "Experiments with program locality," in *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*. ACM, 1972, pp. 611–621.

[13] R. B. Bunt and J. M. Murphy, "The measurement of locality and the behaviour of programs," *The computer journal*, vol. 27, no. 3, pp. 238–245, 1984.

[14] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th annual symposium on Computer Architecture*. IEEE Computer Society Press, 1981, pp. 135–148.

[15] T. Ball and J. R. Larus, *Branch prediction for free*. ACM, 1993, vol. 28.

[16] Y. Liu, L. Guo, P. Liu, and J. Tan, "Compressing regular expressions dfa table by matrix decomposition," *Implementation and Application of Automata*, pp. 282–289, 2011.

[17] M. Becchi, "regex tool," http://regex.wustl.edu/.