PiDFA: A Practical Multi-stride Regular Expression Matching Engine Based On FPGA

Jiajia Yang, Lei Jiang*, Qiu Tang, Qiong Dai, Jianlong Tan
Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
University of Chinese Academy of Sciences, Beijing, China

***Abstract*—DPI technology has been widely deployed in networking intrusion detection system (NIDS) to detect attacks or viruses. State-of-the-art NIDS uses deterministic finite automata (DFA) algorithms to perform regular expression matching for its stable matching speed. However, traditional DFA algorithm's throughput is limited by the input character's width (usually one character per time). Although the multi-stride method (process multiple characters per time) can increase the throughput, it leads the DFA transition table to an exponentially increased memory consumption. In this paper, we propose a novel multi-stride regular expression matching engine called PiDFA based on Field-Programmable Gate Array (FPGA). It applies two methods to solve traditional multi-stride algorithms' memory explosion problem: DFA Transition Merging method and top-k state extraction method. Experiment results show that PiDFA achieves more than 30-fold better performance than original DFA algorithm. Whats more, PiDFA is orthogonal to existing transition table compression algorithms. Implemented with PiDFA algorithm, ClusterFA's matching speed is increased by 6-50 times while maintaining ClusterFA's low memory consumption.**

***Keywords*—Regular Expression Matching; Deep Packet Inspection; DFA; FPGA**

I. Introduction

Regular expression (regex) plays an important role on modern networking intrusion detection systems (NIDSs). It is widely used in state-of-the-art systems for detecting intrusion and virus attacks, including L7-filter [1] and Snort [2]. Usually, regex rules can be converted to non-deterministic finite automata (NFA) and DFA. Many systems implement regex matching engine based on DFA algorithm for its guaranteed worst-case performance ($O(1)$ time per character). However, DFA algorithms suffer from the state blowup problem. Many previous works have been presented to reduce the memory consumption of DFA algorithm by using the compression mechanism. Although compression mechanism is an effective way to reduce memory consumption, it results in low throughput. Because compression mechanism leads to multiple states' traverse processing per input character. The multi-stride method is one way to increase the regex matching engine's performance by processing multiple input characters per time. However, it is not feasible to implement in NIDS practically for its huge memory requirement.

To solve multi-stride algorithms' memory explosion problem, in this paper we present a novel regex matching accelerating method called PiDFA (parallel-input DFA). PiDFA takes the advantage of the parallelism of FPGA to accelerate regex matching by inputting multiple characters per time. Through consuming k characters per time in pipeline, PiDFA yields a k-fold performance improvement than original DFA algorithm. We apply two methods in PiDFA: DFA Transition Merging (DFA-TM) method and top-k state extraction method.

The DFA-TM takes advantage of the parallelism and pipeline of FPGA hardware. It is influenced by the idea of "Divide and Conquer" and is somehow like the method of "Merge Sort". Firstly, the PiDFA engine reads k characters in parallel , then transmits the k characters to the transition merging module. The transition merging module constructs temporary merging transition tables according the input characters. When finishing merging, PiDFA picks out the right results from the last merging transition table. Since the method of DFA-TM run in pipeline, PiDFA can achieve high performance.

The top-k state extraction method is another important technique in PiDFA. If processing large amount of states, it is hardly to implement PiDFA on FPGA effectively for the extremely complex routing and layout problem. For example, when processing 1024 states for 4-character input DFA, we spent more than one week to construct the corresponding PiDFA on a server with 32-core CPU and 64GB memory space. The top-k method utilizes the locality of DFA transition table that there are only a few states be traversed frequently when scanning a sample of real world traffic. We extract the few most frequent states and their corresponding transitions to implement by the DFA-TM method. Then we store the less-frequently states and their corresponding transitions into the BlockRAM of FPGA. In this way, we greatly simplify the layout complexity of PiDFA when implementing on FPGAs.

Beyond that, PiDFA is orthogonal to the DFA transition table compression algorithms. We significantly improve existing DFA compression algorithm's performance by applying PiDFA to it. In this paper, we choose the ClusterFA algorithm to modify and get a better throughput by one order of magnitude.

In particular, the contributions of this paper are summarized as follows:

(1) We propose the PiDFA algorithm to accelerate the regular expressing matching speed. By the DFA-TM method and the top-k state extraction method, PiDFA yields 6-50 times performance increase than original DFA algorithm.
(2) We apply the PiDFA scheme to the DFA transition table compression algorithm. Implementation on ClusterFA show that PiDFA is orthogonal to existing DFA compression algorithms. The modified ClusterFA algorithm achieves 30-fold performance im-

*Corresponding author: Email: jianglei@iie.ac.cn

978-1-4799-6664-6/16/$31.00 ©2016 IEEE

provement while remaining ClusterFA's low memory consumption.

The rest of this paper is organized as follows. Section II discusses the previous work related to regular expression matching. Section III presents the detail of PiDFA. Section IV describes the complete architecture of PiDFA engine implemented on FPGA. Section V analyzes the experiment results of PiDFA. Finally, section VI concludes this paper.

II. Related Work

Regex matching is initially studied as a topic in automaton theory and formal theory in the context of theoretical computer science [3]. Recently, many works have been proposed to promote regex matching, especially DFA algorithm's practical application in NIDS. Currently, researches on regex matching mainly focuses on two aspects: One is to reduce the memory consumption of regular expression automaton, the other is to improve the throughput of regular expression matching engine.

There are a lot of compression strategies to reduce the memory consumption of regex transition table. Kumar et al. [4] observe the similarity of transitions between two states and applying default transitions to compress DFA transition table. At the expense of accessing memory multiple times per input character, D^{2}FA reduces transitions by more than 95% compared to original DFA. Becchi et al. in [5] propose a DFA compression algorithm, called A-DFA. By quantifying a states distance from the initial state, A-DFA results in at most $2N$ state traversals when processing a string of length N. In comparison to the D^{2}FA method, A-DFA yields a comparable compression ratio and has lower complexity. QI Y et al. [6] propose a solution named FEACAN for front-end acceleration of content-aware network processing. FEACAN uses a two-dimensional DFA compression algorithm to reduce the memory consumption and designs a hardware lookup engine to enhance the matching speed. L. Jiang et al. [7] present a new structure, called ClusterFA. This algorithm uses the clustering algorithms to cluster the similar states together and calculate a common state for each cluster. In this way, ClusterFA achieves memory consumption by more than 95%.

Besides, some works focus on improving the performance of regex matching engines. Dharmapurikar et al. [8] present a bloom-filer method to optimize the regex automata. By consuming multiple input characters per time, this algorithm achieves higher throughput than original DFA with moderate memory consumption. Brodie et al.[9] use Multi-stride DFAs to increase the throughput of regex matching. Specially, a stride-k DFA consumes k characters per state transition, thus yielding a k-fold performance increase. However, Multi-stride DFAs lead to an exponentially increased memory requirement in the worst case. In order to solve the huge memory consumption of k-DFA, Michela Becchi et al.[10] use the methods of alphabet-reduction and default transition compression to compress the k-DFA. What's more, their method can avoid troublesome memory size requirements during constructing the automaton. Experimental results show that more than 800 complex regexes can be implemented on an FPGA. But using compression algorithms to compress k-DFA leads to the performance of k-DFA decrease. Y. Liu et al. [11] propose the NFA-OBDDs algorithm to efficiently process sets of NFA frontier states by using ordered binary decision diagrams. By stride doubling, NFA-OBDD can extends to k-stride NFA-OBDD easily. However, NFA-OBDDs only conduct experiments for $k = 2$. Because it is limited by the memory consumption for larger values of k.

	a	b	c	d	e
0	0	0	2	1	2
1	0	1	0	0	1
2	2	2	1	1	1

Fig. 1. DFA transition table: the state-set is {1,2,3} and the alphabet set is {a,b,c,d,e}.

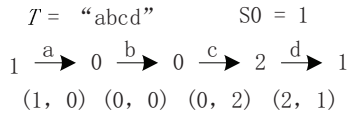

Fig. 2. Traditional DFA process input characters in sequence: the initial state $S0$ = 1, the byte stream T = "$abcd$".

The multi-stride method is widely used in previous works to improve the performance of regular expression matching engines. However, suffering from the state explosion problem, the multi-stride method is usually used as a complimentary approach, and is hardly implemented in practical applications. In the following sections, we will propose a practical k-stride regular expression matching algorithm, getting a high performance while retaining efficiency in memory consumption.

III. DETAIL OF PiDFA

This section is divided into two parts. Part I describes the method of DFA-TM and discusses its benefits and weakness. Part II describes the model and the lookup procedure of PiDFA.

A. Part I : the DFA-TM method

1) An example for traditional DFA lookup

In order to describe the lookup procedure of PiDFA, traditional DFA lookup method that processes input characters in sequence is given firstly. Regex rules can be compiled into a DFA transition table, as shown in Fig.1. The DFA transition table has two dimensions. One is the alphabet dimension and the other is the state dimension. Supposing the initial state $S0$ was equal to 1. If given a byte stream T = "$abcd$", the lookup procedure can be illustrated in Fig.2. The initial state $S0$ traverses to state 0 after consuming character 'a' of T. And then state 0 traverses to itself after consuming character 'b'. After consuming character 'c', the state 0 traverses to state 2. In the end, state 2 traverses to state 1 after consuming character 'd'. The lookup procedure is represented as $(1,0)(0,0)(0,2)(2,1)$. In another word, $(1,0)$ represents state 1 traverses to state 0. $(0,0)$ represents state 0 traverses to state 0. $(0,2)$ represents state 0 traverses to state 2. The last bracket $(2,1)$ represents state 2 traverses to state 1. So the traverse path of T = "$abcd$" is $(1,0)(0,0)(0,2)(2,1)$.

2) An example to explain the method of DFA-TM

The example to explain DFA-TM is a dynamic lookup process. It uses parallel characteristic of FPGA to accelerate the lookup process. There are 7 boxes in total: I, II, III, IV, V, VI, VII. In every box, the first column in red color is

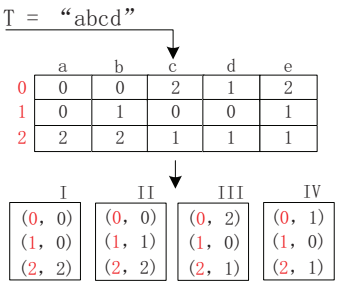

Fig. 3. Get the corresponding columns from DFA transition table by T ("*abcd*"), and fill them into corresponding box.

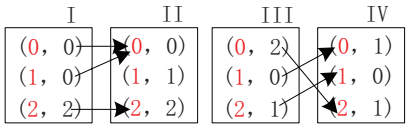

Fig. 4. The box I merging transitions with the box II; the box III merging transitions with the box IV.

the ascending sequence numbers, which are the same as the sequence numbers of states in the DFA transition table in Fig.1. Now we input a byte stream T = "*abcd*". Then we search the DFA transition table by each character of T and select the corresponding columns from DFA transition table to fill in box I, II, III, V respectively, as shown in Fig.3. For example, we use character 'a' to select the first column from DFA transition table, and then we fill the ascending sequence numbers and the selected column into box I. The other characters 'b' , 'c' and 'd' do this procedure by the same way. In order to facilitate explaining our follow-up work, we add the brackets for each row. In each pair of brackets, the left is the sequence number, and the right is the outgoing transition of each character corresponding to the column in the DFA transition table.

Next, as shown in Fig.4, we use the box I to connect to the box II, and the box III to connect to the box IV. The black arrow represents *connecting operator*. It means binding two rows into a row. For example, by binding the 1-th row of the

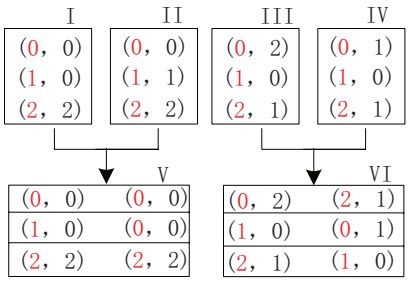

Fig. 5. Merging transitions between the box I and the box II, then put the results into the box V; merging transitions between the box III and the box IV, then put the results into the box VI.

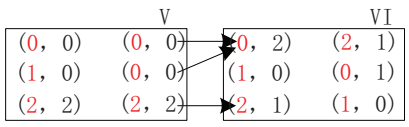

Fig. 6. The box V merging transitions with the box VI.

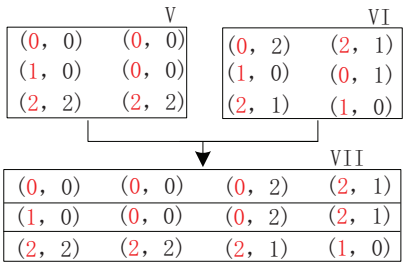

Fig. 7. Merging transitions between the box V and the box VI, then put the results into the box VII.

box I (i.e. $(1,0)$) and the 0-th row of the box II (i.e. $(0,0)$), we get a new row $(1,0)(0,0)$. Now we explain how to connect the box I to the box II. *Connecting operator* must follow this principle: the i-th brackets of the box I (i.e. the i-th row of the box I) uses its right value to connect to the j-th brackets of the box II (i.e. the j-th row of the box II), whose left sequence number is equal to the right value of the i-th brackets of the box I. After connecting, we put the new row into the i-th row in the box (i.e. box V in Fig.5). For example, we use the 0-th row of box I (i.e. $(0,0)$) to connect to the 0-row of box II (i.e. $(0,0)$), as the right value of the 0-th row of box I is equal to the left sequence number of the 0-th row of box II. After connecting, putting the new row $(0,0)(0,0)$ into the 0-th row of the box V. Concurrently, the 1-th row $(1,0)$ of the box I is connected to 0-th row $(0,0)$ of the box II, as the right value of the brackets $(1,0)$ is equal to the left sequence number of the 0-th row of the box II. After connecting, we put the result $(1,0)(0,0)$ into the 1-th row of box V. In the same way, the box III is connected to the box IV and the result is put into box VI. Notice that because of the parallelism of FPGA, the box I and the box II get the results at the same time. All these *connecting operators* just take one clock cycle in parallel. Through repeating this process, the box V is connected to the box VI and the result will be put into the box VII in the end. This procedure is somehow similar to the method of Merge Sort which influenced by the idea of Divide and Conquer. So we call this method DFA transition merging.

After the box VII gets the result, a module, named "Validate Module" (VM), picks out the correct path in the box VII with the information of current state. If the current state is 'i', the VM would select the i-th row. For example, if the current state is 1, the VM selects the 1-th row of the box VII. That is $(1,0)(0,0)(0,2)(2,1)$. Obviously, the result is the same as processed by traditional DFA lookup method in Fig.2 , which proves the correctness of DFA-TM. Now we know that when inputting characters "*abcd*" by order, the lookup process will traverse state 0,0,2,1 by order. Thus, we can justify whether a rule is matched, since the information about the lookup process is known.

3) Benefits and weakness of DFA-TM

Benefits: We can use parallelism and pipeline to accelerate this process. Thus we can consume 4 characters per time. **What's more, we can scale up the number of boxes to process more characters per time**.

Supposing S represents the number of states, W represents the max number of characters consuming per time. The number of ASCII characters is $N = 256$. Then the memory consump-

tion is equal to formula (1), where $M1 = S * log_2S * 256$ represents the size of original DFA transition table and $M2 = S * log_2S * W * 2 * (log_2W + 1)$ represents the logic resources consumed by the process of DFA-TM. We express the ratio of M to M1 as ρ.

$$M = M1 + M2 = S * log_2S * 256 + S * log_2S * W * 2 * (log_2W + 1) \quad (1)$$

$$\rho = M/M1 = 1 + W * (log_2W + 1)/128 \quad (2)$$

Assuming that $W = 4$, then $\rho = 1 + 0.09375$. It means that the method of DFA-TM just only consumes 9.38% logic resources more than the original DFA transitions table, while it can achieve 4-fold speed matching performance than the traditional DFA lookup method. The method of DFA-TM is also much more efficiency than the stride doubling algorithms which have been proposed in [9][10](i.e. k-DFA named in [9]), since the memory consumed by the k-DFA is exponentially increasing compared with the original DFA transitions table. That is $M3 \approx S * log_2S * N^W$. It leads to the k-DFA can't be effectively implemented on the current chip with limited capacity. Although the traditional compression algorithms can reduce the memory consumption, it has little effect on the k-DFA. What's worse, compression algorithms result in performance decrease. From the formula (2) ,it infers that the memory consumption of our method is $O(Wlog_2W)$ increase while k-DFA is $O(N^W)$ increase. Thus the method of DFA-TM is much more efficiency than the k-DFA.

Weakness: The disadvantage of DFA-TM is that it can only be implemented by using logic resources (e.g. flip-flop (FF) and look-up table (LUT)) but not memory at present. If the number of states is too large, the compiler tool can't $Place\&route$ (a stage of Xilinx ISE tool).

B. Part II: the model and the process of PiDFA

As discussing in part I, if the number of states is too large, the compiler tool can't implement our method effectively. In order to solve this problem, we reduce the states implemented by logic resources. We extract the few most frequent states (top-k states) to be implemented by the method of DFA-TM and store the rest of DFA transition table into BlockRAM, which does not include the most frequent states and their corresponding transitions.

1) Locality

Lots of previous works in the area of DFA matching has been proposed [12][13]. In these works, the common idea is that a bulk of the DFA transitions are concentrated around a few DFA states. We call this phenomenon as "$locality$". The measure results of the $locality$ are shown in Fig.8.

Data sets are publicly available at MIT Lincoln Lab [14], named $directory$ (3.0MB), $fs\_listing$ (6.6MB), $hume\_evt$ (4.3MB), $inside$ (161.2MB), $outside$ (155.7MB) and $pascal$ (5.4MB). The regex rules are all publicly available real-life rule sets, such as Bro, Snort and L7-filter. Here we just present the results measured by data set $outside$.

In Fig.8, top-k represents the k most frequent states. From this figure, we get more than 99% frequency access on a few most frequent states most of the time. In the experiment, we

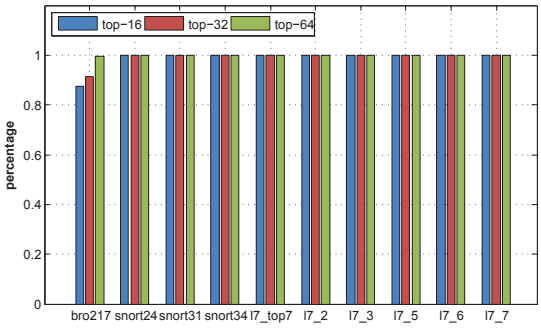

Fig. 8. Locality measured by data set $outside$ of MIT Lincoln Lab.

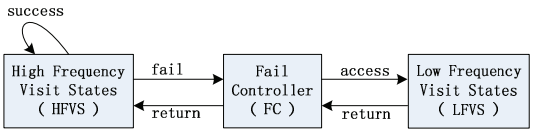

Fig. 9. Model of PiDFA: DFA transition table is divided into HFVS and LFVS.

find two phenomenons: *1)* a bulk of the DFA transitions are concentrated around a few DFA states of every rule set; *2)* different rule sets have different most frequent states. From Fig.8, it implies if the number of the most frequent states is greater than 64, then almost all of the transitions will always traverse among these states except for bro217.

2) Model of PiDFA

The model of PiDFA is shown in Fig.9. We extract the most frequent states and store them into the "High Frequency Visit States" (HFVS) module. The rest states are stored into the "Low Frequency Visit States" (LFVS) module. Measured by real cases, most of the input characters can be processed by the HFVS. If failed, the process will jump to the "Fail Controller" (FC) module to access the LFVS. After finishing matching, the process will return to the HFVS and continue the next round process.

3) Lookup procedure for PiDFA

We reorganize the DFA transition table and divide it into HFVS and LVFS. Then we add flag bits for each column of HFVS, just as shown in Fig.10 and Fig.11. The flag bit represents whether the value of corresponding state is out of range of the states in the HFVS. If the value is larger than the max sequence number of states, corresponding flag bit is set

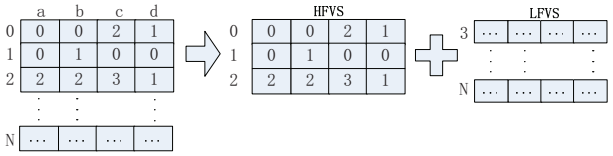

Fig. 10. Divide DFA transition table into HFVS and LFVS.

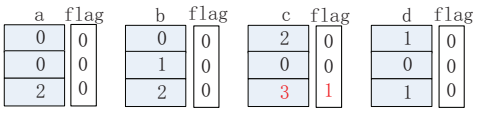

Fig. 11. Add flag bits for each column of the HFVS.

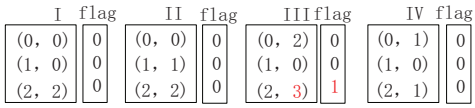

Fig. 12. Get the corresponding columns and corresponding flag bits from DFA transition table and the box of flag bits by T ("$abcd$"), and fill them into corresponding box.

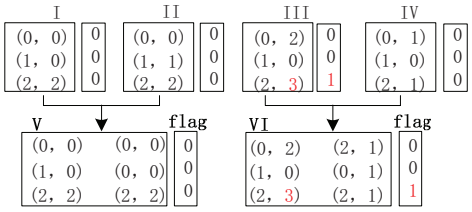

Fig. 13. Merging transitions between the box I and the box II, then put the results into the box V; merging transitions between the box III and the box IV, then put the results into the box VI.

to 1. Otherwise it is set to 0. For example, as shown in Fig.11, the 2-th row of the box c is 3. That is larger than the max state sequence number of HFVS (the max sequence number of states is 2). So the corresponding flag bit is set to 1 in the corresponding flag box.

In Fig.12, the 2-th row of the box III is $(2, 3)$. The corresponding flag bit is 1 and the right value is 3. The flag bit implies that 3 is larger than the max sequence number of states in the HFVS. So we just put the 2-th row of the box III and the 2-th row of the box IV into the box VI, and set the corresponding flag bit of the 2-th row of box VI to 1. The rest rows of the box III and the rest rows of the box VI are processed by the same way as the process of DFA-TM, as shown in Fig.13.

Repeating this process, the final connecting results are shown in Fig.14. Then the VM selects the correct result with the current state. For example, if the current state is 0, VM selects the row $(0, 0)(0, 0)(0, 2)(2, 1)$. But if the current state is 2, VM selects the row $(2, 2)(2, 2)(2, 3)(2, 1)$. It checks out the flag bit at the same time and finds that the corresponding flag bit is 1. That means in this row, there must be some values larger than the max sequence number of states in HFVS. So VM quits the row $(2, 2)(2, 2)(2, 3)(2, 1)$ and uses the byte stream T = "$abcd$" to access the LFVS. Similarly, if current state is larger than 2, then VM will directly use the byte stream

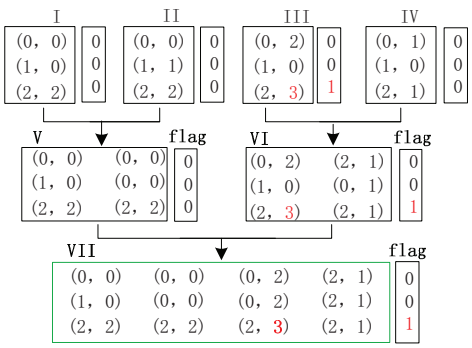

Fig. 14. Merging transitions between the box V and the box VI, then put the results into the box VII.

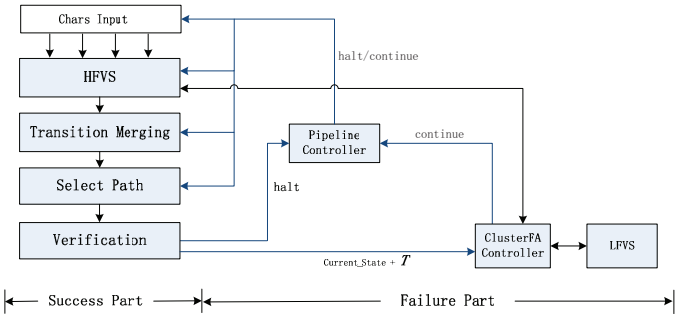

Fig. 15. PiDFA engine with two parts: Success Part and Failure Part.

T to access the LFVS.

IV. HARDWARE IMPLEMENTATION

A. Design an engine for PiDFA

In order to reduce the memory consumption, we use compression algorithms to work with our PiDFA. The PiDFA engine consists of 8 parts: Chars Input Module (CIM), H-FVS, Transition Merging Module (TMM), Select Path Module (SPM), Verification Module (VM), Pipeline Controller (PCM), Compression Algorithm Controller Module (CACM), LFVS, as shown in Fig.16. The five parts of Success Part run in pipeline if there is not any state's value larger than the max sequence number of states in HFVS during validated by the VM. Otherwise, VM sends a signal "$halt$" to the PCM. After receiving the signal, the PCM notice the CIM, SPM and TMM to keep in the current context. Concurrently, the byte stream T, which are stored in a buffer (not show in this figure), and the current state will be sent to the CACM to access between the LFVS and the HFVS. When the lookup procedure finishes, the LFVS stores the next state into the buffer. Meanwhile, it sends a signal "$continue$" to the PCM. This signal notices the CIM, SPM, and TMM to continue to work at next round process. Since most input characters are proccessed in HFVS, the LFVS could not be accessed at most time. As a result, the PiDFA can achieve high throughput.

B. Benefits and weakness of PiDFA

1) Benefits of PiDFA

As discussing in section III, only a few most frequent states are implemented by LUT and FF. The rest states of DFA transitions table are stored in BlockRAM. Supposing S represents the number of states. H_S represents the most frequent states. L_S represents the less frequent states. W represents the size characters of T (i.e. the input characters per time). The number of ASCII characters is $N = 256$. Thus we get these formulas as follows.

$$M_L = L_S * log_2 S * 256 \tag{3}$$

$$M_H = H_S * log_2 S * 256 + H_S * log_2 S * W * 2 * (log_2 W + 1) \tag{4}$$

$$M_f lag = H_S * 1 * 256 + H_S * (2W - 1) \tag{5}$$

$$\rho = (M_L + M_H + M_{flag})/(S * log_2 S * 256) \tag{6}$$

Where M_L represents the size of LFVS implemented by BlockRAM and M_H represents the logic resources consuming by Success Part. M_{flag} represents the size of flag bits. Thus the ratio of the size of original DFA transition table to M_L is shown as formula (6). If S $\gg H_S$ and W is not much too large, then $\rho \approx 1$. That means PiDFA just consumes a small amount of LUT and FF resources more than the original DFA transition table, while it can consume W input characters per time.

V. Performance Evaluation

The experiments are performed on an Ubuntu 12.4 operating system (CPU: i5-3470 Core, 3.20 GHz; Memory: 8G). The evaluation tool is Xilinx ISE 14.7. Hardware simulation is based on a Xilinx Virtex-7 FPGA chip ($XC7VX690T$) with $693,120$ logic cells (LCs), and $52,920Kb$ BlockRAM.

The traditional DFA, the D^{2}FA (restrict the depth of D^{2}FA to 1) and the ClusterFA are mainly implemented by BlockRAM (i.e. it consumes little logic resources), while the PiDFA is implemented by LUT, FF and BlockRAM together. Therefore, we separately evaluate the usage of LUT and FF consumed by PiDFA, and then we compare the PiDFA with other algorithms in terms of BlockRAM consumption.

We set the H_S equal to 64 (i.e. top-k is 64) and get the regex rules from Bro, Snort and L7-filter. Data sets are publicly available at MIT Lincoln Lab, as described in section III. Besides these data sets, we also capture 1.5GB raw packets (i.e. raw_data) from our network. We use the data sets of $directory$, $fs\_listing$, $inside$ and $outside$ to extract the most frequent states, while using $hume\_evt$, $pascal$ and raw_data to measure throughput. In addition, we use the ClusterFA algorithm to compress the states stored in LFVS. We call this method Pi-ClusterFA or PIC. Then using 2PI, 4PI, 8PI, 16PI, and 32PI represents 2, 4, 8, 16, and 32 input characters per time for the PiDFA respectively. In the same way, 2PIC, 4PIC, 8PIC, 16PIC and 32PIC represents 2, 4, 8, 16, and 32 input characters per time for the Pi-ClusterFA respectively. For the exponentially increased memory requirement, we don't treat the k-DFA as an experimental object. Because it can't be implemented on the Xilinx Virtex-7 FPGA chip effectively.

1) speedup

We use $hume\_evt$, $pascal$ and raw_data to measure the speedup. The values shown in TABLE I are the average throughput of different algorithms. The best performance is $29.59Gbps$, which is near 30-fold of the original DFA. Compared with DFA, D^{2}FA and ClusterFA algorithms, the PiDFA shows better performance.

However, PiDFA has different effect on different rule sets. As shown in TABLE I, the performance of rule set bro217 is lower than other rule sets. This is mainly caused by the characters which need to access LFVS. Take 32PI for example, when characters keep being processed in the Sucess Stage, it just takes 1 step to handle 32 characters. But if there is a character need to access LFVS, it takes 32 steps to handle 32 characters. So the more characters need to access LFVS, the lower throughput of the PiDFA is. Besides, the performance of PiDFA is better than the Pi-ClusterFA as shown in TABLE II. The unit of throughput is $Gbps$.

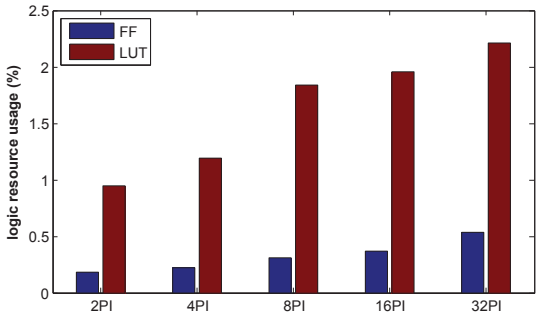

Fig. 16. LUT and FF consumption for PiDFA.

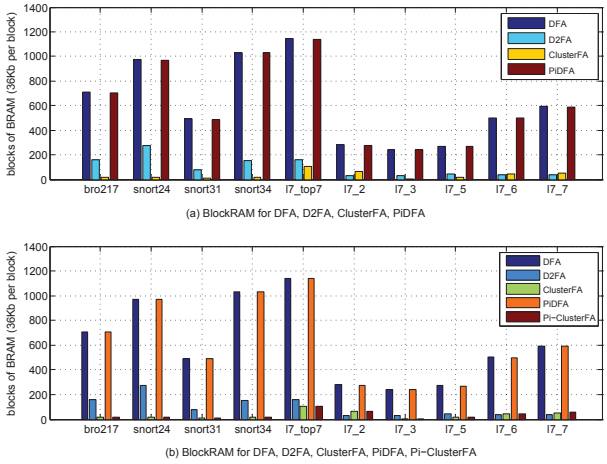

Fig. 17. BlockRAM consumption for different algorithms.

2) Memory consumption

From Fig.16 , we can see that the PiDFA only consumes a few FF and LUT resources of the whole logic resources of Virtex-7 FPGA chip (the total FFs is 866400 and the total LUTs is 433200). The consumption of logic resources is no more than 2.3% of the total logic resources.

The total memory resources is 1470 blocks of BlockRAM. The size of each block is $36Kb$. The PiDFA is compared with other algorithms in Fig.17. The Fig.17(a) shows the amount of BlockRAM consumed by the PiDFA is almost the same as the original DFA. The Pi-ClusterFA is compared with other algorithm as shown in Fig.17(b). From Fig.17(b) , we can see that the Pi-ClusterFA consumes less BlockRAM than other algorithms in most rule sets. It implies that our PiDFA is orthogonal to some other compression algorithms well.

VI. Conclusion

Traditional multi-stride DFA algorithm is limited by the exponentially increased memory requirement. In this paper, we improve the multi-stride algorithm by two methods: 1) the DFA Transition Merging method and 2) the top-k state extraction method. The PiDFA algorithm effectively solves the state explosion problem led by the multiple characters input. In the experiments, we use some real-life regular expression rule sets from Snorts, Bro and L7-filter. The results show that PiDFA achieves more than 30-fold better performance than original DFA algorithm. What's more, we validate the orthogonality

TABLE I. THROUGHPUT OF DIFFERENT ALGORITHMS

rulesets	state No.	DFA	D^2FA	ClusterFA	2PI	4PI	8PI	16PI	32PI
bro217	6941	1.16	1.02	0.67	**2.33**	**4.32**	**7.14**	**8.64**	**5.70**
snort24	8577	1.15	1.08	0.53	**2.31**	**4.53**	**8.75**	**16.72**	**29.59**
snort31	4806	1.17	0.61	0.51	**2.32**	**4.53**	**8.75**	**16.74**	**29.46**
snort34	10194	1.12	0.67	0.51	**2.31**	**4.51**	**8.68**	**16.71**	**29.31**
l7_top7	11218	1.08	1.01	0.51	**2.34**	**4.49**	**8.74**	**16.71**	**29.34**
l7_2	2732	1.08	1.04	0.65	**2.34**	**4.50**	**8.74**	**16.71**	**29.48**
l7_3	2342	1.08	1.05	0.69	**2.31**	**4.51**	**8.75**	**16.74**	**29.48**
l7_5	2647	1.08	1.03	0.69	**2.33**	**4.51**	**8.75**	**16.74**	**29.48**
l7_6	4913	1.08	1.04	0.68	**2.33**	**4.50**	**8.75**	**16.72**	**29.47**
l7_7	5818	1.08	1.04	0.69	**2.33**	**4.51**	**8.75**	**16.72**	**29.46**

TABLE II. COMPARISON BETWEEN THE THROUGHPUT OF PIDFA(PI) AND PI-CLUSTERFA(PIC)

rulesets	state No.	4PI	8PI	16PI	32PI	4PIC	8PIC	16PIC	32PIC
bro217	6941	4.32	7.14	8.64	5.70	4.10	5.29	4.92	3.39
snort24	8577	4.53	8.75	16.72	29.59	4.41	8.24	15.83	27.82
snort31	4806	4.53	8.75	16.74	29.46	4.41	8.24	15.96	27.93
snort34	10194	4.51	8.68	16.71	29.31	4.38	8.16	15.70	27.61
l7_top7	11218	4.49	8.74	16.71	29.34	4.24	8.07	15.59	27.51
l7_2	2732	4.50	8.74	16.71	29.48	4.37	8.16	15.70	27.60
l7_3	2342	4.51	8.75	16.74	29.48	4.38	8.17	15.76	27.75
l7_5	2647	4.51	8.75	16.74	29.48	4.38	8.17	15.74	27.71
l7_6	4913	4.50	8.75	16.72	29.47	4.36	8.14	15.70	27.61
l7_7	5818	4.51	8.75	16.72	29.46	4.37	8.17	15.70	27.61

of PiDFA and existing DFA compression algorithms. Implemented with PiDFA algorithm, ClusterFA's matching speed is increased by 6-50 times while maintaining ClusterFA's low memory consumption. In addition, it must be pointed out that the multi-stride DFA matching algorithms is meaningful to the design of routers. In state-of-the-art router's design, the datapath's width is usually more than 64 bits. It is a great challenge to add the regular expression matching engine to the main datapath of routers. Next,we will implement a practical 64-bit input PiDFA module and plug it into NetFPGA design [15].

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China under Grant No. 61402475, and Xinjiang Uygur Autonomous Region Science and Technology Project under Grant No. 201230123.

REFERENCES

[1] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *LISA*, vol. 99, no. 1, 1999, pp. 229–238.

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

[3] J. E. Hopcroft, *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[4] 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.

[5] M. Becchi and P. Crowley, "A-dfa: a time-and space-efficient dfa compression algorithm for fast regular expression evaluation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, p. 4, 2013.

[6] 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.

[7] L. Jiang, J. Tan, and Y. Liu, "Clusterfa: a memory-efficient dfa structure for network intrusion detection," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 2012, pp. 65–66.

[8] S. Dharmapurikar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *Selected Areas in Communications, IEEE Journal on*, vol. 24, no. 10, pp. 1781–1792, 2006.

[9] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2. IEEE Computer Society, 2006, pp. 191–202.

[10] M. Becchi and P. Crowley, "Efficient regular expression evaluation: theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2008, pp. 50–59.

[11] L. Yang, R. Karim, V. Ganapathy, and R. Smith, "Fast, memory-efficient regular expression matching with nfa-obdds," *Computer Networks*, vol. 55, no. 15, pp. 3376–3393, 2011.

[12] L. Jiang, Q. Dai, Q. Tang, J. Tan, and B. Fang, "A fast regular expression matching engine for nids applying prediction scheme," in *Computers and Communication (ISCC), 2014 IEEE Symposium on*. IEEE, 2014, pp. 1–7.

[13] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Multi-byte regular expression matching with speculation," in *Recent Advances in Intrusion Detection*. Springer, 2009, pp. 284–303.

[14] https://www.ll.mit.edu/ideval/data/.

[15] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "Netfpga–an open platform for gigabit-rate network switching and routing," in *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*. IEEE, 2007, pp. 160–161.