High Performance Regular Expression Matching on FPGA

Jiajia Yang, Lei Jiang(✉), Xu Bai, and Qiong Dai

School of Cyber Security, Institute of Information Engineering,
University of Chinese Academy of Sciences, UCAS,
Beijing, People's Republic of China
{yangjiajia,jianglei,baixu,daiqiong}@iie.ac.cn

Abstract. Deep Packet Inspection (DPI) technology has been widely deployed in Network Intrusion Detection System (NIDS) to detect attacks and viruses. State-of-the-art NIDS uses Deterministic Finite Automata (DFA) 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). In this paper, we present an architecture named Parallel-DFA to accelerate regular expression matching by scanning multiple characters per time. Experimental results show that, our architecture can achieve as high as 1200 Gbps (1.17 Tbps) rate on current single Field-Programmable Gate Array (FPGA) chip. This makes it a very practical solution for NIDS in 100G Ethernet standard network, which is currently the fastest approved standard of Ethernet. To the best of our knowledge, this is the fastest matching performance architecture on a single FPGA chip. Besides, the throughput is nearly 3 orders of magnitude (916×) than that of original DFA implemented on software. Our architecture is about 183.2× efficiency than that of original DFA.

Keywords: Deep Packet Inspection · Regular expression matching
DFA · FPGA · Network security

1 Introduction

Networking security in collaborative networks is becoming crucial with the various attacks networks are being exposed to [1]. In order to protect networking devices and computer systems from such attacks, Signature-based DPI technologies have taken root as a dominant security mechanism to detect and neutralize potential threats by discarding malicious traffic. Most of the attacks are described by regular expressions (REs) for its powerful and flexible expressive ability. There has been a lot of recent works on implementing REs for use in high-speed networking environment, particularly with representations based on DFA [2]. However, DFA suffers from huge memory consumption. Moreover, matching network traffic against a DFA is inherently a serial activity. These existing

I. Romdhani et al. (Eds.): CollaborateCom 2017, LNICST 252, pp. 541–553, 2018.
https://doi.org/10.1007/978-3-030-00916-8_50

problems make DFA a challenge to be implemented in current limited resources network devices.

In order to accelerate DFA matching, many accelerated methods on different platforms have been proposed. These methods have already achieved high throughput theoretically but still suffer their drawbacks respectively. Some schemes achieve high throughput at the expense of consuming huge memory, making them impractical in current dedicated hardware with limited resources. Thus, satisfying low memory consumption and high throughput requirement in high-speed network environment simultaneously is difficult and challenging.

In this paper, we present Parallel-DFA, a high-throughput DFA accelerated architecture. Using tight integration of logic resources available on-chip on FPGA, we can achieve a throughput as high as 1200 Gbps. With the requirements of high performance in high-speed network and the advancements in memory technology, coping with the acceleration problem has become a firstly concern while the memory efficiency is a secondly concern. Thus, Parallel-DFA mainly focuses on the acceleration of DFA. We summarize our contributions as follows:

(1) A high-speed DFA accelerated architecture on a single chip. It's throughput can achieve as high as 1200 Gbps.
(2) Theoretical analysis of time and space consumption of our proposed architecture. We also compare the time and space consumption of our architecture with that of original DFA and multi-stride DFA.
(3) Detailed performance analysis wrt throughput, resource usage and the number of supported rules under various configurations.

The rest of this paper is organized as follows. Section 2 discusses the previous work related to RE matching. Section 3 describes the detail of Parallel-DFA architecture. Section 4 describes the performance evaluation. Finally, Sect. 5 concludes this paper.

2 Related Work

With the widespread use of REs, RE matching has become research interests and a hot topic in academic and industries area. Many works have been extensively studied to promote RE matching, especially DFA algorithms' practical application in NIDS. Currently, researches on RE matching mainly focus on two aspects: One is to reduce the memory consumption of RE automaton, the other is to improve the throughput of RE matching.

Many compression strategies have been proposed to reduce the memory consumption of RE transition tables. Most of the strategies relate to compress techniques based on the characteristic of RE transition tables. Kumar et al. [3] propose D^2FA based on observing 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^2FA achieves more than 95% compression ratio than the original DFA transition table. Motivated by the idea of D^2FA, Becchi et al. in [4] propose A-DFA, which is also

a novel DFA compression algorithm. A-DFA quantifies a state's distance from the initial state, and results in at most $2N$ state traversals when processing an input string of length N. In comparison with the D^{2}FA method, A-DFA yields a comparable compression ratio and has lower complexity. Jiang et al. [5] use the clustering algorithms to cluster the similar states together. In this way, they achieves memory consumption by more than 95%.

Besides, some strategies focus on improving the performance of RE matching. For example, Brodie et al. [6] use multi-stride DFAs to increase the throughput. 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. Jan et al. [7] present a pattern-matching engine built on an extended version of the B-FSM scheme that provides a substantially more compact data structure on DFA. Experimental results show the B-FSM implemented on FPGA can achieve scan rates on 16 Gb/s per B-FSM. Chad [8] et al. use dedicated TCAM to accelerate RE matching. In order to reduce TCAM space and improve matching speed, three novel techniques are proposed: transition sharing, table consolidation, and variable striding. Their method can achieve potential matching throughput of 10 to 19 Gbps using only a single 2.36 Mb TCAM chip. But it is challenging to implement RE on TCAM for its expensive and not scalable with respect to clock rates and power consumption.

3 The Detail of Our Architecture

This section includes 3 subsections. Subsection 3.1 introduces original DFA and multi-stride DFA. Subsection 3.2 gives the description of our algorithm. In Subsect. 3.3, we design and analyze our hardware architecture.

3.1 The Introduction of Original DFA and Multi-stride DFA

We introduce the basic original DFA by an example. Figure 1 (I) shows a standard DFA defined on the alphabet a, b. In this DFA, state 0, 1 (in white cycle) are normal states, while state 2 (in grey cycle) is match state. We define a graph $G = <V, E>$, where V is a set of vertices (states) and E is a set of edges (transitions). In this example, $V = \{v_0, v_1, v_1\} = \{0, 1, 2\}$, where v_i is a vertex. $E = \{e_0, e_1, e_2, e_3, e_4, e_5\}$, where e_i denotes a transition (edge). A transition denotes a state (vertex) jumps to another state. Its additional information is an alphabet, such as a, b. If the initial state is state 0 and a input string is '*abab*', the traversal path will be: $(0) \xrightarrow{a} (1) \xrightarrow{b} (2) \xrightarrow{a} (0) \xrightarrow{b} (0)$.

Suppose the length of *input string* is n and the number of DFA states is S. As the number of ASCII characters is 256, original DFA graph's space consumption is: $256 \times S \times \lceil log_2 S \rceil$, where $\lceil log_2 S \rceil$ denotes the bit-width of an edge (transition). The processing time of original DFA is $O(n)$.

Because the processing time is too large for DFA, Becchi et al. [9] introduce classical multi-striding technique, which is used to increase throughput of a system for matching strings against a set of REs. An example is given in Fig. 1 (II).

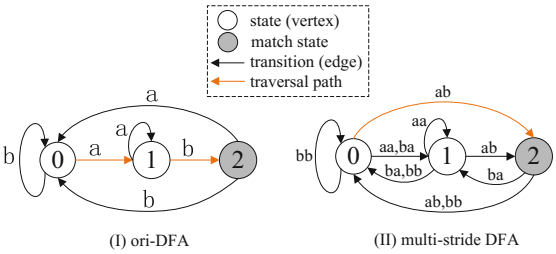


Fig. 1. The DFA algorithms. (I) the original DFA. (II) a multi-stride DFA that consumes 2 characters per time.

In this 2-DFA, it can consume 2 characters per time. If processing n characters, it just consumes $\frac{n}{2}$ times. In another words, it can achieve 2× speedup of original DFA. The traversal path is: $(0) \xrightarrow{ab} (2) \xrightarrow{ab} (0)$.

As for multi-stride DFA, its space consumption is about $S \times \lceil log_2 S \rceil \times 256^n$. This is a huge memory consumption. Although the time complexity is only $\frac{1}{2}$ of original DFA's, the space consumption is too huge to make it practical on current limited resource device.

3.2 The Description of Our Algorithm

Now, we give an example to show the process of our algorithm in Fig. 2. In Fig. 2, our algorithm can be divided into four stages: **(1)** we split original DFA into 4 subgraphs based on the input string 'abab'. The additional information of

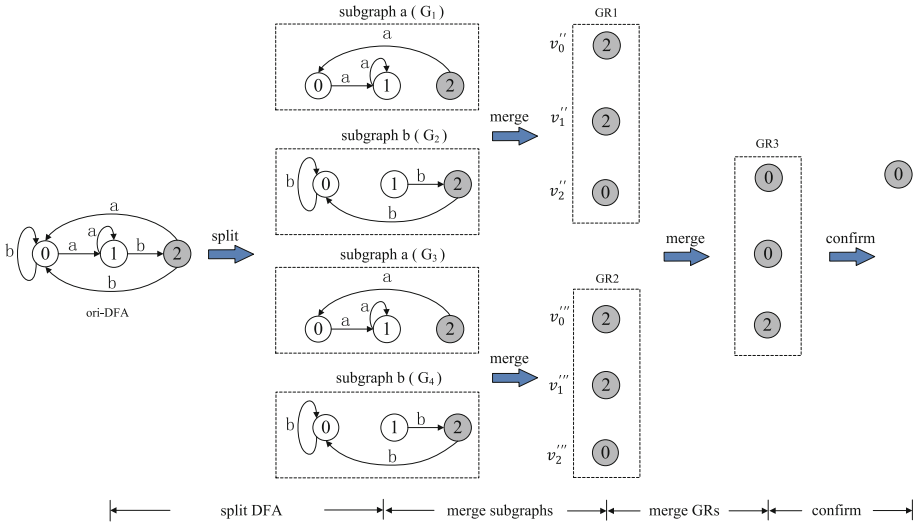


Fig. 2. An example to show the whole process.

Algorithm 1. MergeSubGraph(G_j, G_k)

Input: G_j, G_k
Output: G
Procedures:
1: $G.V \leftarrow \varnothing$, $G.E \leftarrow \varnothing$
2: **for all** v, v', ($v \in G_j.V$, $v' \in G_k.V$) **do in parallel**
3: **if** $\exists$ v_{mid}, satisfy $<v,v_{mid}> \in G_j.E$, $<v_{mid},v'> \in G_k.E$ **then**
4: **if** these states v,v_{mid},v' has one in grey **then**
5: v' is in grey
6: **end if**
7: $G.V \leftarrow v'$
8: **end if**
9: **end for**

Algorithm 2. MergeNodeGroup(GR_x, GR_y)

Input: GR_x, GR_y
Output: GR_z
Procedures:
1: $GR_z.V \leftarrow \varnothing$, $GR_z.E \leftarrow \varnothing$
2: **for all** $v_i^{''}$, $v_j^{'''}$, ($v_i^{''} \in GR_x.V$, $v_j^{'''} \in GR_y.V$) **do in parallel**
3: **if** $\exists$ i,j, satisfy $j == v_i^{''}$ **then**
4: $GR_z \leftarrow v_j^{'''}$
5: **end if**
6: **end for**

transitions of each subgraphs is only an alphabet. **(2)** We merge these subgraphs to GRs (i.e. a group of state nodes). **(3)** We merge GRs into the last GR. **(4)** If given the *current state*, we confirm the *next state*, and justify whether the input string 'abab' is accepted by this DFA.

The pseudo-code of stage (2) is shown in Algorithm 1. By Algorithm 1, we can merge two subgraphs G_j, G_k into a GR, which only contains state nodes. The pseudo-code of stage (3) is shown in Algorithm 2. By Algorithm 2, we can merge two GRs into one GR.

We give an example to explain the Algorithm 2. In Fig. 2, $GR_1 = \{v_0^{''}, v_1^{''}, v_2^{''}\} = (2,2,0)$, where $v_0^{''} = 2$, $v_1^{''} = 2$, $v_2^{''} = 0$. $GR_2 = \{v_0^{'''}, v_1^{'''}, v_2^{'''}\} = (2,2,0)$, where $v_0^{'''} = 2$, $v_1^{'''} = 2$, $v_2^{'''} = 0$. Because $v_0^{''} = 2$, then $GR_3.V \leftarrow v_2^{'''}$; $v_1^{''} = 2$, then $GR_3.V \leftarrow v_2^{'''}$; $v_2^{''} = 0$, then $GR_3.V \leftarrow v_0^{'''}$. In summary, $GR_3 = \{v_2^{'''}, v_2^{'''}, v_0^{'''}\} = (0,0,2)$.

The pseudo-code of the whole process is shown in Algorithm 3. Line 1–11 show how to split original DFA. The ori-DFA is split into several subgraphs based on input string str. $\{G\}$ denotes a set of subgraphs. Line 6–8 show all the edges that their additional information equals to the input character will be extracted to form a subgraph. Line 12–17 show how to merge these subgraphs. Each two adjacent subgraphs are merged into a GR. All the $GR^{'}$s will be put into the set $\{GR\}$. Line 18–26 show how to merge these GRs. Line 21–23 show

Algorithm 3. The whole process

Input: G, str, the *current state*
Output: *accepted*
Procedures:

```
1: // 1.split DFA based on input string str
2: i ← 0
3: {G} ← ∅
4: for all character ch ∈ str do in parallel
5:    ∀ e ∈ DFA //e := edge
6:    if (e.character == ch) then
7:       Copy e and its vertices to G_i // form the i^th subgraph
8:    end if
9:    {G} ← {G} ⋃ G_i
10:   i++
11: end for
12: // 2.merge subgraphs
13: {GR} ← ∅
14: for all G_2i,G_2i+1 (G_2i,G_2i+1 ∈ {G}) do in parallel
15:   GR' ← MergeSubGraph(G_2i,G_2i+1)
16:   {GR} ← {GR} ⋃ GR'
17: end for
18: // 3.merge GRs
19: while ({GR} only has a group) do
20:   for all GR_2i,GR_2i+1 (GR_2i,GR_2i+1 ∈ {GR}) do
21:      GR_new ← MergeNodeGroup(GR_2i,GR_2i+1)
22:      { } ← { } ⋃ GR_new
23:   end for
24:   {GR} ← ∅
25:   {GR} ← { }
26: end while
27: // 4. justify whether the input string is accepted
28: GR_last ← {GR}
29: if current state == i then
30:   next state = GR_last.v_i
31:   if (GR_last.v_i is in grey) then
32:      accepted = Y // Y := yes
33:   else
34:      accepted = N // N := no
35:   end if
36: end if
```

we merge each two adjacent GRs into a GR, then put it into a new set { }. After merging, all the GR_{new}s in { } are put into previous $\{GR\}$, which has been clear out (Line 24–25). Finally, we only has a GR in $\{GR\}$ (line 19). Line 27–36 show whether the input string str is accepted. As $\{GR\}$ only has a group (i.e. GR), we set $GR_{last} = GR$. If $GR_{last}.v_i$ is in grey cycle, then str is accepted (line 32). Otherwise, it's not accepted (line 34).

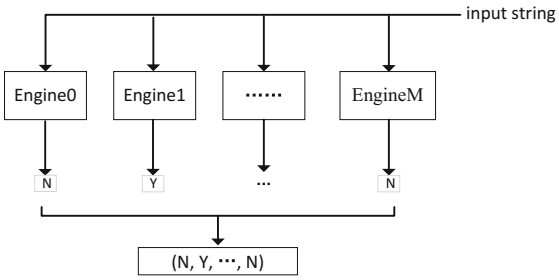


Fig. 3. Each rule is compiled into an engine. Our architecture consists of multiple engines.

The space consumption of these subgraphs is: $n \times S \times \lceil log_2 S \rceil$. The space consumption of GRs is: $S \times \lceil log_2 S \rceil \times (n-1)$. Space consumption mainly includes the consumption of DFA, subgraphs and GRs. So our algorithm's total memory consumption is: $(2n - 1 + 256) \times S \times \lceil log_2 S \rceil$. In general, S is a constant, so the space complexity of algorithm 3 is $O(n)$. If algorithm 3 is implemented by non-pipelined technique, it takes $(\lceil log_2(n) \rceil + 1)$ clock cycles to process n characters. If accelerated by the pipelined technique, it only takes 1 clock cycles to process n characters.

3.3 Hardware Architecture

(1) *design our hardware architecture*
Suppose each rule is compiled into an engine, which performs RE matching by Algorithm 3. Since a ruleset may include many rules, we make these engines in parallel. An example is given in Fig. 3. In Fig. 3, the $Engine_i$ denotes the i^{th} engine. The lookup process is fairly simple. The *input string* is sent into these engines. At last, each engine shows whether the *input string* is accepted.

(2) *the analysis of space and time consumption*
Suppose there are m rules. The i^{th} rule is compiled into S_i states. The length of input string is n. So the total space consumption of our architecture is: $\sum_{i=1}^{m}[(2n-1)+256] \times S_i \times \lceil log_2 S_i \rceil$. If we use original DFA as baseline, the ratio of space consumption between our architecture and original DFA is shown in Eq. (1):

$$MA = \frac{\sum_{i=1}^{m}[(2n-1)+256] \times S_i \times \lceil log_2 S_i \rceil}{\sum_{i=1}^{m} 256 \times S_i \times \lceil log_2 S_i \rceil} = \frac{2n+255}{256} \tag{1}$$

Suppose $m = 4$, $S_i = 16$, $n = 8$, then $MA = 1.06$. That means we only consume additional 6% space consumption than the original DFA's, while we can achieve 8× speedup acceleration than that of original DFA.

The ratio of space consumption between multi-stride DFA and original DFA is shown in Eq. (2):

$$MA \approx \frac{\sum_{i=1}^{m} \times S_i \times \lceil log_2 S_i \rceil \times 256^n}{\sum_{i=1}^{m} 256 \times S_i \times \lceil log_2 S_i \rceil} = 256^{n-1} \tag{2}$$

The ratio of time consumption between our pipelined architecture and original DFA is shown in Eq. (3). The ratio of time consumption between multi-stride DFA and original DFA is also same as Eq. (3):

$$TA = \frac{1}{n} \tag{3}$$

From Eq. (3), our architecture's time consumption is equal to the multi-stride DFA. However, the multi-stride DFA's space consumption is too huge to be implemented practically. Suppose $n = 8$, the $MA = 256^7 = 2^{56} \gg 1.06$.

Theoretically, we have the following inequation (4), where C denotes the available resource and k denotes coefficient of utilization.

$$\sum_{i=1}^{m} [(2n-1) + 256] \times S_i \times \lceil log_2 S_i \rceil \leq k \times C. \tag{4}$$

We change the expression form of inequation (4)–(5).

$$n \leq \frac{1}{2} \times (\frac{k \times C}{\sum_{i=1}^{m} S_i \times \lceil log_2 S_i \rceil} - 255) \tag{5}$$

We use $freq$ to denote the evaluated frequency of FPGA. As for pipelined architecture, the throughput is shown in Eq. (6), while the throughput of non-pipelined architecture is shown in Eq. (7).

$$TH_{pipelined} = freq \times n \times 8 \tag{6}$$

$$TH_{non-pipelined} = \frac{freq \times n \times 8}{\lceil log_2(n) \rceil + 1} \tag{7}$$

We plug Eqs. (5)–(6) and get Eq. (8). Meanwhile, we plug Eqs. (5)–(7) and get Eq. (9).

$$TH_{pipelined} \leq 4 \times freq \times (\frac{k \times C}{\sum_{i=1}^{m} S_i \times \lceil log_2 S_i \rceil} - 255) \tag{8}$$

$$TH_{non-pipelined} \leq 4 \times freq \times (\frac{k \times C}{\sum_{i=1}^{m} S_i \times \lceil log_2 S_i \rceil} - 255) \times \frac{1}{\lceil log_2(n) \rceil + 1} \tag{9}$$

Now, we set $k = 0.7$, which is the (maximum) average value of Fig. 5 calculated by inequation (4). Also, we set $freq = 300\,\text{MHz}$ depended on our experimental synthetical frequency. We set $C = 712000$, which equals to the total available resource of our experimental FPGA chip.

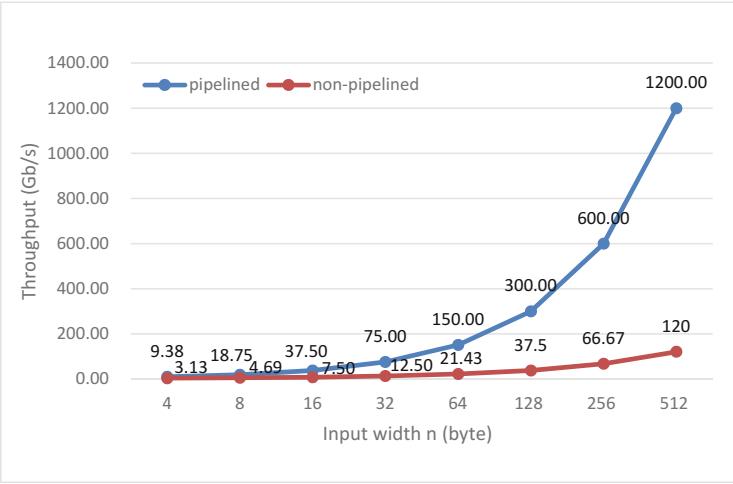


Fig. 4. The throughput of pipelined and non-pipelined architecture.

Suppose each S_i is the same. In the case of $m = 1$, $S_i = 32$, we get $n \leq 1430$ (byte). The $TH_{pipelined} \leq 3.27\,(\text{Tb/s})$, $TH_{non-pipelined} \leq 272.75\,(\text{Gb/s})$. In another words, we can get the maximum throughout $MAX(TH_{pipelined}) \approx 3.27\,(\text{Tb/s})$, $MAX(TH_{non-pipelined}) \approx 272.75\,(\text{Gb/s})$. However, the throughput of 3.27 Tb/s is hard to achieve by the place-and-route of Xilinx tool.

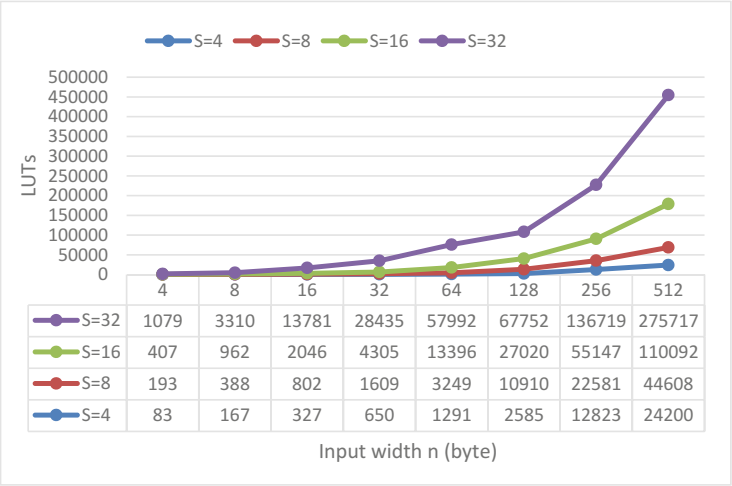
4 Performance Evaluation

In this section, we provide a detailed analysis of our architecture on a state-of-the-art Xilinx chip XC7VX1140t. This considered chip has 712000 LUTs. The RE rules are all publicly available in real-life rulesets, such as Bro [11], Snort [12]. The performance is measured in throughput, resource consumption and efficiency. However, since these rules are finally compiled into DFA states, we use states as a measurement standard but not the rules themselves.

4.1 Throughput

Synthesis was performed by using the Xilinx tool. From the synthetical result, we use 300 MHz to evaluate the throughput. We can deduce the throughput based on the considered frequency. The input widths (i.e. the lengths of input string) are rounded up to be a power of two.

Our architecture can be implemented on two forms: pipelined-architecture and non-pipelined-architecture. Figure 4 shows the throughput variation with the size for various input width. n denotes the input width. It can be observed that our proposed pipelined-architecture can sustains 1200 Gbps, while the non-pipelined-architecture only sustains 120 Gbps at the best case. It can be seen



	4	8	16	32	64	128	256	512
S=32	1079	3310	13781	28435	57992	67752	136719	275717
S=16	407	962	2046	4305	13396	27020	55147	110092
S=8	193	388	802	1609	3249	10910	22581	44608
S=4	83	167	327	650	1291	2585	12823	24200

Fig. 5. Under the conditions of different number of states Ss, the LUT's consumption vs. the input width n.

that on the average, using a larger input width size if desirable from a throughput point of view. With the increasing of input width, the throughput is also increasing. Apparently, our architecture is practical for 100G Ethernet standard that is currently the fastest approved standard of Ethernet. Noted that because of stable performance, the throughput is independent of tested data.

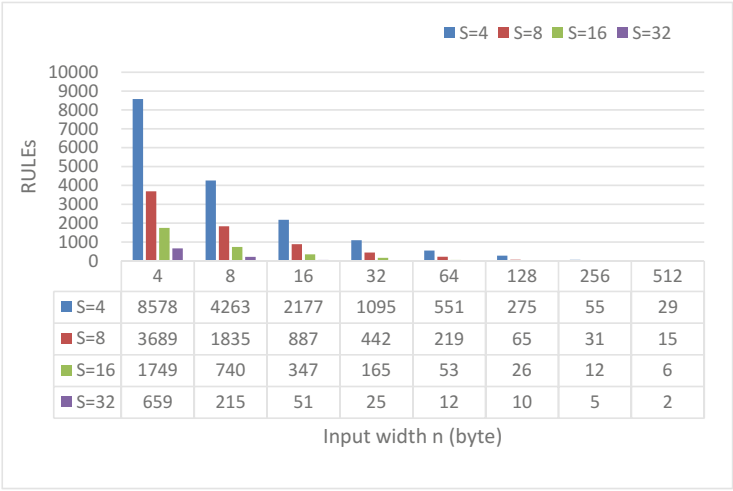
4.2 Resource Consumption

The resource consumptions of pipelined and non-pipelined architecture are almost the same. In this work, only logic resources are utilized. Figure 5 illustrates the LUT's consumption of each rule. Suppose S denotes the number of states each rule generated. We can observer that: (1) under the conditions of the same number of states, with the increasing of input width, the consumption of LUT is increasing. (2) under the conditions of the same input widths, with the increasing of the number of states, the consumption of LUT is also increasing. (3) the larger value of the number of states S and input width n, the much more consumption of LUTs. In the case of $n = 512$ and $S = 32$, almost $275717/712000 = 38.72\%$ of the logic resources available on this considered FPGA are consumed.

Besides, that how many rules can be implemented in parallel is shown in Fig. 6. In the best case of $n = 4$ and $S = 4$, our single chip can support 8578 rules. In the case of $n = 512$ and $S = 32$, this chip only supports 2 rules.

4.3 Comparison with Existing Approaches

In order to make a fair comparison, we set $S = 32$. The evaluated frequency of FPGA is 300 MHz. We compare throughput and space consumption between



	4	8	16	32	64	128	256	512
S=4	8578	4263	2177	1095	551	275	55	29
S=8	3689	1835	887	442	219	65	31	15
S=16	1749	740	347	165	53	26	12	6
S=32	659	215	51	25	12	10	5	2

Fig. 6. The number of rules supported by our current chip under the conditions of various Ss and ns.

Table 1. Comparison between different algorithms and platforms.

Approach	Throughput (Gbps)	Space consumption (byte/trans.)	Efficiency (Gbps * trans./byte)	Platform	Stable
Software DFA	1.31	1.63	0.80	Intel CPU	No
TCAM [8]	10–19	N/A	N/A	TCAM	No
B-FSM [7]	18.4	1.38	13.33	PowerEn	No
GregEx [13]	25.6	1.63	15.71	GPU	No
Multi-DFA	4.69	417.28	0.01	FPGA	Yes
Ours	1200	8.80	136.36	FPGA	Yes

*Software-based DFA was carried out on a dual core, 2.93 GHz Intel(R) Core(TM) matchine with 8 GB of RAM and running Ubuntu Linux 12.4.

different DFA algorithms or platforms. Multi-stride DFA's *stride* is equal to 2 for resource of this chip is limited. Although compression algorithms lead to memory efficiency, they result in low throughput. So we doesn't implement compression algorithms on it.

We compare space consumption between different platforms. Table 1 summarizes these comparisons. The throughput of our architecture is about 1200 Gbps, which is 916 times than that of original software-based DFA. Though space consumption is not the best, our architecture's efficiency can achieve as high as 136.36.

5 Conclusion

We propose an algorithm to accelerate regular expression matching. Based on this algorithm, we present Parallel-DFA, a regular expression matching architecture. It supports 1200 Gbps throughput in our experiment on current Xilinx Virtex-7 chip. What's more, its performance is stable. It must be pointed that our architecture's throughput is independent of any tested data. These advantages make it suitable for modern high-speed collaborative network security environment. In the future, with more resources available on FPGA, we can enlarge the value of input width easily to support higher throughput.

Acknowledgment. Supported by the National Science and Technology Major Project under Grant No. 2017YFB0803003, the National Science Foundation of China (NSFC) under grant No. 61402475.

References

1. Dubrawsky, I.: Firewall evolution-deep packet inspection. In: Security Focus, vol. 29 (2003)
2. Hopcroft, J.E.: Introduction to Automata Theory, Languages, and Computation. Pearson Education India (1979)
3. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. ACM SIGCOMM Comput. Commun. Rev. **36**(4), 339–350 (2006)
4. Becchi, M., Crowley, P.: A-DFA: a time-and space-efficient DFA compression algorithm for fast regular expression evaluation. ACM Trans. Arch. Code Optim. (TACO) **10**(1), 4 (2013)
5. Jiang, L., Dai, Q., Tang, Q., Tan, J., Fang, B.: A fast regular expression matching engine for NIDS applying prediction scheme. In: IEEE Symposium on Computers and Communication (ISCC), pp. 1–7. IEEE (2014)
6. Brodie, B.C., Taylor, D.E., Cytron, R.K.: A scalable architecture for high-throughput regular-expression pattern matching. ACM SIGARCH Comput. Arch. News **34**(2), 191–202 (2006)
7. Van Lunteren, J., Rohrer, J., Atasu, K., Hagleitner, C.: Regular expression acceleration at multiple tens of Gb/s. In: 1st Workshop on Accelerators for High-performance Architectures in conjunction with ICS 2009 (2009)
8. Meiners, C.R., Patel, J., Norige, E., Liu, A.X., Torng, E.: Fast regular expression matching using small TCAM. IEEE/ACM Trans. Netw. (TON) **22**(1), 94–109 (2014)
9. Becchi, M., Crowley, P.: Efficient regular expression evaluation: theory to practice. In: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 50–59. ACM (2008)
10. Prithi, S.. Sumathi, S.: Review on grouping algorithms for finite state automata (2016)
11. The Bro Network Security Monitor. http://www.bro.org

12. Roesch, M.: Snort: lightweight intrusion detection for networks. LISA **99**(1), 229–238 (1999)
13. Wang, L., Chen, S., Tang, Y., Su, J.: Gregex: GPU based high speed regular expression matching engine. In: 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), pp. 366–370. IEEE (2011)