A Matching Method for Look-ahead Assertion on Pattern Independent Regular Expression Matching Engine

Yoichi Wakaba Shinobu Nagayama Masato Inagi Shin'ichi Wakabayashi

Graduate School of Information Sciences
Hiroshima City University
3-4-1 Ozuka-higashi, Asaminami-ku, Hiroshima, 731-3194 Japan
waka27@lcs.info.hiroshima-cu.ac.jp {s_naga, inagi, wakaba}@hiroshima-cu.ac.jp

Abstract— In this paper, we propose a matching method for look-ahead assertions for pattern in regular expression matching. In network intrusion detection systems (NIDSs), look-ahead assertions are often used to compactly describe patterns to find strings that do not include specified substrings. However, as far as we know, existing pattern independent matching engines which can handle look-ahead assertions have not been proposed. In the proposed matching method, we introduce a preprocessing circuit into the pattern independent matching engine which we have previously proposed. It performs matching for look-ahead assertions by searching from the end of a text to the beginning of the text. To realize a high throughput, we also propose a new buffer organization using stack memory which is suitable for the proposed engine. Experimental results show that the proposed engine handles look-ahead assertions efficiently.

I. Introduction

Network security is becoming more and more important in today's information-oriented society. Network intrusion detection systems (NIDSs), which monitor network traffic and notify users of detections of malicious traffic, are indispensable to improve network security. NIDSs perform pattern matching between predefined regular expression patterns and packet payloads to find malicious traffic. If a packet payload includes a predefined pattern, alert message is sent to administrations.

NIDSs have been commonly realized by software, such as Snort [1]. Patterns for Snort, called Snort rules, are described in extended regular expression using several specific operators for compactly describing patterns. Since the speed of a network is improved rapidly and regular expression matching is a very intensive and time-consuming task, real time regular expression matching by software becomes difficult.

In recent years, a number of hardware engines for regular expression matching have been proposed [2–7]. Most of them implement a non-deterministic finite automaton (NFA) corresponding to given patterns on a reconfigurable device, such as FPGA [2–5]. The engines can realize fast regular expression matching with compact circuit size, but they cannot update patterns immediately. This is because architectures of the engines depend on the given patterns, and they need re-synthesis and reconfiguration of circuits whenever patterns for NIDSs are updated. It can be a significant bottleneck for NIDSs, in which patterns are frequently updated.

To perform fast regular expression matching and quick pattern updating, pattern-independent hardware matching engines, based on a systolic algorithm [6] and on an NFA [7], have been proposed. While the NFA based engine is faster than the systolic engine, the NFA based engine cannot handle the larger class of patterns than the systolic one can handle. The systolic based engine can handle most patterns in Snort rules. Thus, in this paper, we focus on the systolic based engine. However, the systolic based engine proposed in [6] cannot handle patterns including look-ahead assertions.

In NIDSs, look-ahead assertions are often used, for example, to detect emails from unregistered domain names. Such a pattern is described using a look-ahead assertion as follows: ".*@(?! *hiroshima-cu\.ac\.jp*)". This matches email addresses whose domain names are other than "@hiroshima-cu.ac.jp". Although a pattern with look-ahead assertions could be translated into a pattern using only basic operations of regular expression, it produces a too long pattern requiring a huge amount of hardware resources to implement a matching engine and long translation time. Therefore, a method that can directly deal with look-ahead assertions using only reasonable hardware resources is desirable.

In this paper, we propose a matching method for look-ahead assertions on the systolic based engine. In the proposed method, we introduce a preprocessing circuit into the engine. It performs matching for look-ahead assertions by searching from the end of a text to the beginning of the text, efficiently using stack memory. Experimental results show that the proposed engine handles look-ahead assertions efficiently.

The rest of this paper is organized as follows. In Section II., we explain regular expression and a systolic based

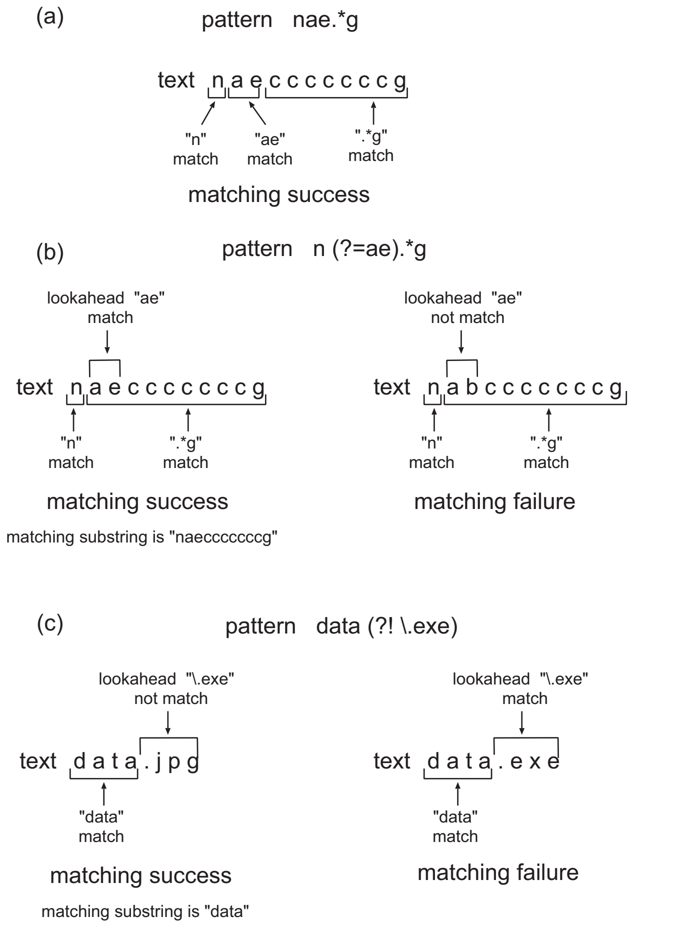

Fig. 1. Examples of matching for look-ahead assertions.

pattern independent matching engine. In Section III., we propose a matching method for look-ahead assertions. In Section IV., we show the experimental results. Finally, the conclusions are presented in Section V.

II. Preliminary

A. Regular Expression

Regular expression is a method to represent a set of strings as a single string with operators. It is often used to describe a pattern compactly. A regular expression represents a set of strings by using three basic operators: union (|), concatenation ($\cdot$) and Kleene operator ($*$). These operators are defined as follows: $R_1|R_2 = \mathrm{L}_1 \cup \mathrm{L}_2$, R_1R_2= $\mathrm{L}_1\mathrm{L}_2$=$\{s_1s_2|s_1 \in \mathrm{L}_1,\ s_2 \in \mathrm{L}_2\}$ and $R^* = \{\varepsilon\} \cup \mathrm{L}^1 \cup \mathrm{L}^2 \cup \cdots$, where ε is a special character which matches the string with no character (*i.e.*, “”), R, R_1 and R_2 are arbitrary regular expressions, L, L_1 and L_2 are sets of strings corresponding to R, R_1 and R_2, respectively.

Extended regular expression can describe a pattern more compactly. It has various operators (e.g., back reference and look-ahead assertions). We explain look-ahead assertions which is important in this paper.

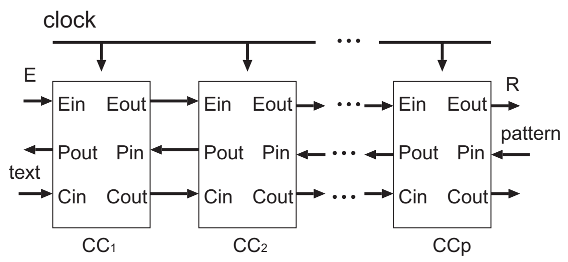

Fig. 2. Architecture of existing matching engine [6]

Look-ahead assertions can be used to compactly describe patterns to match strings that do not include specified substrings, and patterns to match stings that include both of specified two kinds of substrings. It is described as follows: “R_1 (?=R_2) R_3” is positive look-ahead, and “R_1 (?!R_2) R_3” is negative look-ahead, where R_1, R_2 and R_3 are arbitrary regular expressions or may be ‘ε’. Positive look-ahead represents a set of strings that match both R_2 and R_3 immediately after matching R_1. Note that strings that match R_2 are substrings of strings that match R_3, or vice versa. Negative look-ahead represents a set of strings that match R_3 and does not match R_2 immediately after matching R_1. In this paper, a look-ahead assertion denotes a whole pattern “R_1 (?=R_2) R_3” or “R_1 (?!R_2) R_3”, and a look-ahead pattern denotes only R_2.

We show examples of matching for look-ahead assertions in Fig.1. In Fig.1, (b) and (c) are look-ahead assertions, and (a) is not that.

B. Existing Pattern-Independent Matching Engine

As the base of the matching engine proposed in this paper, we use a pattern-independent matching engine based on a systolic algorithm [6]. It is constructed as a one-dimensional array of simple processing units, called comparison cell (CC), as shown in Fig.2. A CC is a synchronous circuit module that performs one-character matching. Each single character in the given pattern is stored in a CC. A pattern is input from the rightmost CC before starting string matching, in advance. A text to be retrieved is input from the leftmost CC, character by character, and one-character matching is performed in each CC in parallel and pipeline fashion. Note that the input directions of a pattern and a text are opposite. Please refer to [6] for more details of the matching engine.

III. The proposed method

In this section, we discuss matching for look-ahead assertions using the systolic based engine [6]. Then, we propose a matching method for look-ahead assertions.

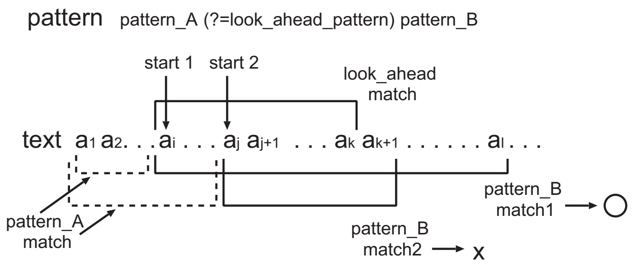
pattern pattern_A (?=look_ahead_pattern) pattern_B
start 1
start 2
look_ahead match
text a1 a2. . . ai . . . aj aj+1 . . . ak ak+1 . . . . . . al . . .
pattern_B match1
pattern_A match
pattern_B match2

Fig. 3. Example of matching for a look-ahead assertion 2.

A. A Straightforward Matching Method for Look-ahead Assertions

If we consider only whether a given pattern matches substrings in a given text or not, that is, if we do not consider where the position of substrings matched a given pattern is, some look-ahead assertions can be handled with the existing engine by converting a look-ahead assertion to a pattern, which uses three basic regular expression operators ("|", "·" and "∗"). For example, a pattern "*abc*(?=\\.*exe*)" can be converted to a pattern "*abc.exe*", and a pattern "*n*(?=*ae*).∗*g*" can be converted to a pattern "*nae.*∗*g*". However, others are difficult and impractical to be converted to regular expression. For example, the pattern "(?! .∗*medical*).∗*virus*" is used, if we want to find emails including word "virus" which have possibility of computer virus, but don't want to find emails including both words "medical" and "virus" which mean virus in medical field. This pattern is impractical to be converted, because the converted pattern is too long and requires a huge matching engine.

One matching method for look-ahead assertions "R_1 (?=R_2) R_3" and "R_1 (?!R_2) R_3" using the systolic based engine would be the one, which performs matching for R_2 and matching for R_3 in parallel. The engine outputs matching success only when matchings of both R_2 and R_3 succeed. In this case, the engine must check whether matching start positions for R_2 and R_3 are same or not, If the matching start positions are different, the engine does not output matching success. For example, we assume that for a look-ahead assertion "*pattern A* (?=*look-ahead pattern*) *pattern B*", "*pattern A*" matches two substrings "$a_1 \cdots a_{i-1}$" and "$a_1 \cdots a_{j-1}$" in text, as shown in Fig.3. In this case, the engine performs matchings of "*look-ahead pattern*" and "*pattern B*" whose start positions are a_i, and also performs matchings of "*look-ahead pattern*" and "*pattern B*" whose start positions are a_j. As shown in Fig.3, since both matchings whose start positions are a_i succeed, the engine must output matching success. On the other hand, when the start positions are a_j, the engine must not output matching success, because the matching of "*look-ahead pattern*" failed. However, if the engine does not distinguish these matchings by their start positions, it would output matching success, since matchings of "*look-ahead pattern*" and "*pattern B*" whose start positions are respectively a_i and a_j succeed. Thus, the engine has to distinguish all matchings performed in parallel by their start positions. Thus, a straightforward method to distinguish all the matchings would be to index all of them. However, this requires too much hardware resources. Therefore, we propose another method.

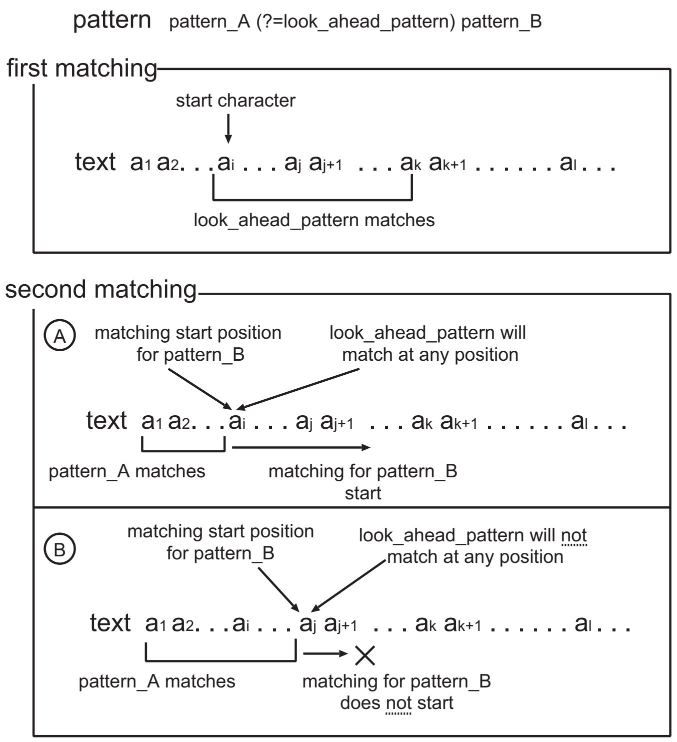
pattern pattern_A (?=look_ahead_pattern) pattern_B
first matching
start character
text a1 a2. . . ai . . . aj aj+1 . . . ak ak+1 . . . . . . al . . .
look_ahead_pattern matches
second matching
A
matching start position for pattern_B
look_ahead_pattern will match at any position
text a1 a2. . . ai . . . aj aj+1 . . . ak ak+1 . . . . . . al . . .
pattern_A matches
matching for pattern_B start
B
matching start position for pattern_B
look_ahead_pattern will not match at any position
text a1 a2. . . ai . . . aj aj+1 . . . ak ak+1 . . . . . . al . . .
pattern_A matches
matching for pattern_B does not start

Fig. 4. The behavior of the proposed matching method

B. The Proposed Method for Look-ahead Assertions

In this subsection, we propose a matching method for look-ahead assertions. The matching engine described in subsection III.A performs matching for a look-ahead pattern and matching for a pattern following the look-ahead pattern in parallel. In the proposed method, matching for a look-ahead pattern is firstly performed, and then matching for a pattern excluding the look-ahead pattern is performed by using the matching result for the look-ahead pattern.

We present the behavior of the proposed matching method in detail using Fig.4. In matching for a look-ahead pattern, the engine finds the start characters of substrings which match the look-ahead pattern by using a method described in subsection III.B-1. In matching for a pattern excluding the look-ahead pattern, if matching start position for *pattern_B* (that is a pattern following the look-ahead pattern) is the start character of substrings which match the look-ahead pattern, the engine performs matching for *pattern_B*. If not, the engine does not perform matching for *pattern_B*.

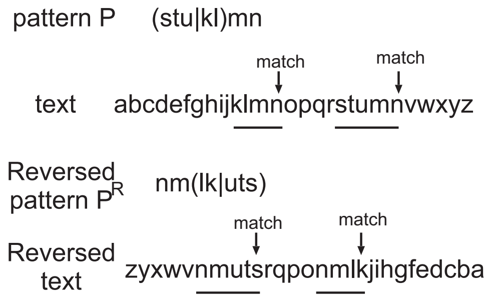

Fig. 5. Matching for a reversed pattern

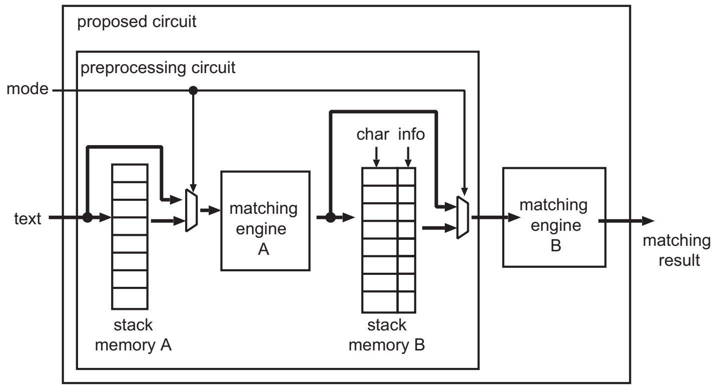

Fig. 6. Architecture

B-1 Searching the start characters of substrings which match a look-ahead pattern

In the proposed method, the matching engine has to find the start characters of substrings which match a look-ahead pattern. However, the matching engine cannot output the start characters, because it detects only the last characters of substrings which match the pattern by searching from the beginning of a text.

Therefore, we give a reversed pattern P^R, in which characters and operators in the original pattern P are ordered reversely, to the engine, and then we also input a text reversely to the engine. In this case, although the engine detects the last characters of substrings which match P^R, those are the start characters of substrings which match P, as shown in Fig.5. In this method, for all substrings Q_i which match P, their reversed substrings Q_i^R must match P^R. But, this is proved in [8].

B-2 Architecture

In this subsection, we present an architecture which realizes the proposed method for matching a look-ahead assertions. The architecture is shown in Fig.6. The architecture consists of the matching engine B based on [6] which performs matching for a pattern excluding a look-ahead pattern and a preprocessing circuit which performs matching for a look-ahead pattern. The preprocessing circuit consists of the matching engine A based on [6] and two stack memories. Note that, if a given pattern does not include a look-ahead assertion, two matching engines, A and B, can be used as one matching engine by directly connecting them. The behavior of the proposed architecture is shown in the following.

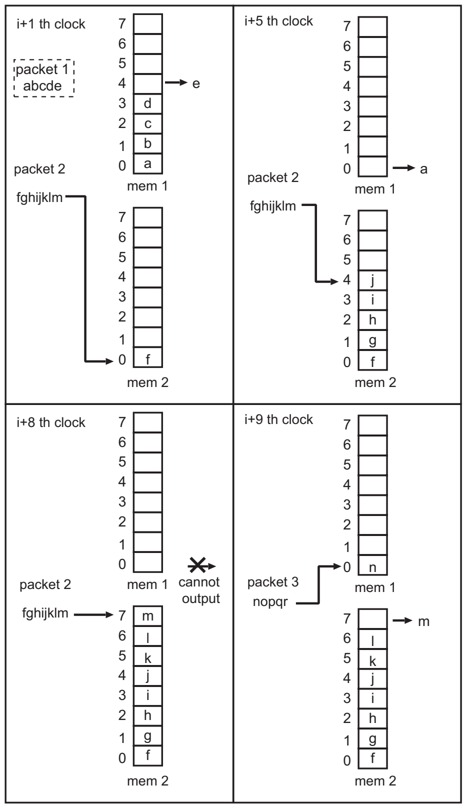

Fig. 7. Stack of packets in double stack memory

1. A reversed look-ahead pattern P^R is set to the matching engine A, and a pattern excluding the look-ahead pattern is set to the matching engine B.
2. When matching starts, a text is stacked in the stack memory A to make a reversed text.
3. After the end character of a text is stacked in the stack memory A, the characters are input to the matching engine A from the end of the text.
4. The matching engine A outputs characters with information about whether the characters are the start characters of substrings which match the look-ahead pattern or not, and they are stacked in the stack memory B.
5. After the beginning character of the text is stacked in the stack memory B, the characters are input to the matching engine B from the beginning.
6. The matching engine B performs matching for the pattern excluding the look-ahead pattern using information generated by the preprocessing circuit.

Note that the proposed method cannot work, when the

Assumption:	The maximum length of packets is n. The size of the proposed memory is $2n$.
Variable:	$\mathit{Write\_address}$. $\mathit{Current\_read\_address}$. $\mathit{Current\_end\_address}$. $\mathit{Next\_start\_address}$. $\mathit{Next\_end\_address}$.
Initialize:	All variables are set to zero.
Step:	
1.	An input packet character is stored to memory [$\mathit{Write\_address}$].
2.	If the input packet character is the last character of a packet, then $\mathit{Next\_start\_address} = \mathit{Write\_address}$.
3.	$\mathit{Write\_address} = \mathit{Write\_address} + 1 \pmod{2n}$.
4.	If $\mathit{Write\_address}$ is n, then go to Step 5. else then go to Step 1.
5.	$\mathit{Current\_read\_address} = \mathit{Next\_start\_address}$.
6.	$\mathit{Current\_end\_address} = \mathit{Next\_end\_address}$.
7.	$\mathit{Next\_end\_address} = \mathit{Next\_start\_address} + 1 \pmod{2n}$.
8.	Output a packet character of memory [$\mathit{Current\_read\_address}$].
9.	An input packet character is stored to memory[$\mathit{Write\_address}$].
10.	If the input packet character is the last character of a packet, then $\mathit{Next\_start\_address} = \mathit{Write\_address}$.
11.	$\mathit{Write\_address} = \mathit{Write\_address} + 1 \pmod{2n}$.
12.	$\mathit{Current\_read\_address} = \mathit{Current\_read\_address}$ - $1 \pmod{2n}$.
13.	If $\mathit{Current\_read\_address}$ is $\mathit{Current\_end\_address}$ - $1 \pmod{2n}$, then go to the Step 5. else, then go to the Step 8.

Fig. 8. The proposed memory algorithm.

length of a given text is infinite. Since our target application is NIDSs, a given text is a packet. The length of a packet is finite and is not extremely long. Therefore, the proposed method is valid for NIDSs.

B-3 The Proposed Stack Memory

In this subsection, we propose a new buffer organization using a stack memory which is suitable for the proposed engine. Throughput of the proposed engine is decreased compared to the previous engine [6], since the proposed engine has to stop working while a given text is stacked in a stack memory. This degradation of throughput is not negligible, because a text is stacked in a memory twice. Note that many texts (*i.e.*, packets) are consecutively input to the proposed engine, since our target application is NIDSs. Hereinafter, we call a given text *a packet*. For example, if the proposed engine uses double buffering of packets, the engine stops working, when the length of an input packet is longer than the length of output packet (Fig.7). Therefore, we propose a new buffer organization using stack memory to reduce degradation of throughput of the proposed engine. If the proposed engine uses the proposed stack memory, the engine stops working only during first n clock cycles, thereafter, the engine will not stop working, where n is the maximum length of packets.

The size of the proposed stack memory is $2n$ and its structure is the same as ring buffer. The dual port memory is required to realize the proposed memory.

The algorithm of the proposed memory is shown in Fig.8. At first, n characters are input to a memory independently of packet sizes. Thus, multiple packets may be stored in the memory. Then, stored packets P_1 except a packet which has not been completely stored yet, begin to be output from the memory. The output order is from the latest input packet to the earliest input packet. While the packets P_1 are output from the memory, new packets P_2 are input to the memory. After all the packets P_1 are output from the memory, the new stored packets P_2 begin to be output from the memory.

Fig.9 shows the behavior of the proposed memory, where the maximum length of packets is 8.

IV. Experimental Evaluation

In this section, we evaluate the area efficiency and the performance of the proposed engines. To evaluate the engines, we used ISE 13.1 as the FPGA design tool and select Xilinx Virtex6 (XC6VLX240T-1FFG1156) as the target device. We implemented the proposed engines for two packet sizes: 1,500bytes, which indicates MTU (Maximum Transmission Unit) for Ethernet and 65,536bytes, which indicates the maximum length of an IP packet.

First, we evaluate the area overhead to realize look-ahead assertions. In our experiments, the existing engine [6] has 20 CCs, and the proposed engine has 10 CCs for the preprocessing circuit and 10 CCs for the matching engine. To estimate the area overhead in terms of the number of slice LUTs of FPGA, we calculate the difference of the numbers of slice LUTs for the proposed engine and the existing engine. The numbers of slice LUTs for the existing engine and the proposed engines supporting packets of 1,500 bytes and 65,536 bytes are 854, 1,017,

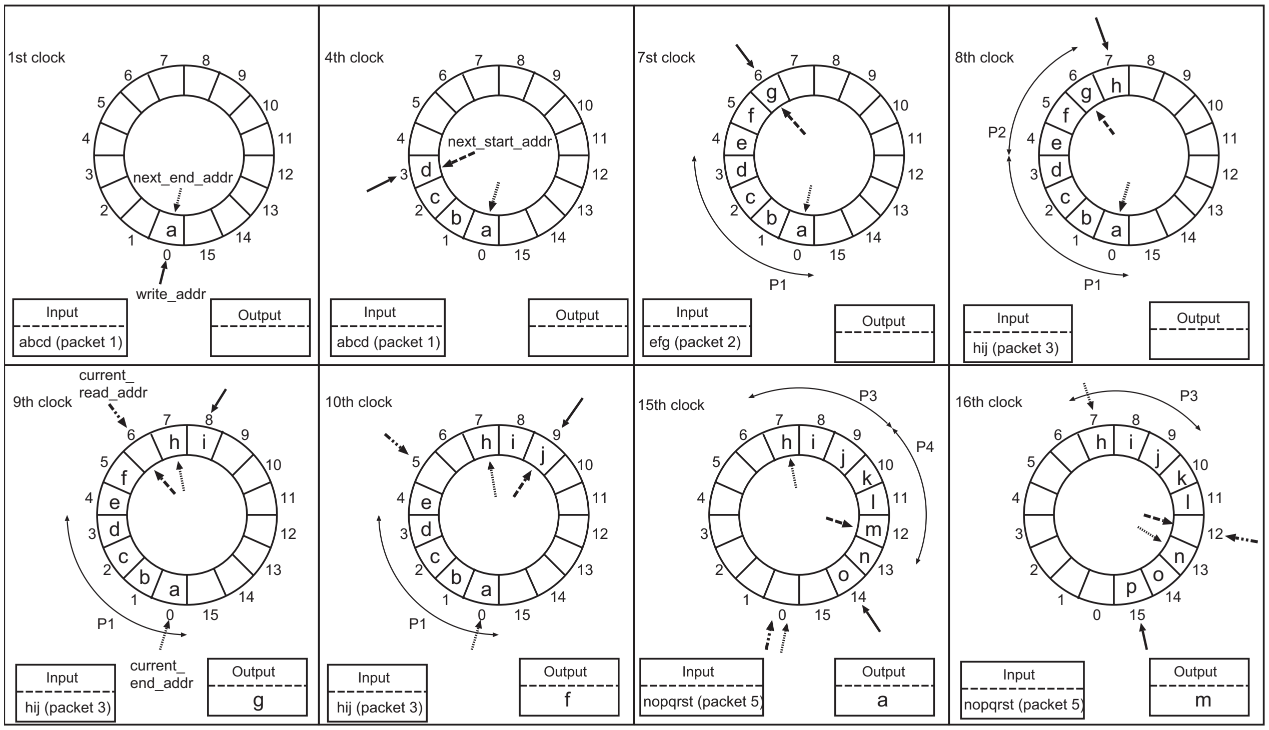

Fig. 9. The behavior of the proposed memory

and 1,293, respectively. Thus, the area overhead is about 163 or 439 slice LUTs. If a look-ahead assertion is converted to regular expression, it requires too many cells of the existing engine resulting in too many slice LUTs. On the other hand, our proposed engines can realize a look-ahead assertion only using slightly more slice LUTs that are comparable to a few cells.

The proposed engine also requires 3 or 92 block RAMs for stack memories. However, this memory overhead is insignificant. This is because the existing engine needs no block RAMs, and stack memories are implemented taking advantage of unused block RAMs inherent in FPGAs. Therefore, the proposed engine can realize look-ahead assertions with only small area overhead.

Next, we evaluate the maximum clock frequency. Those of the existing one and the proposed ones supporting packets of 1,500bytes and 65,536bytes are 294MHz, 251MHz and 145MHz. The performance of the proposed one supporting packets of 65,536bytes degraded greatly because of complex connection of 92 block RAMs to make two stack memories. However, the proposed engines can still detect viruses on Gigabit Ethernet, because their throughputs are more than 1Gbps.

The experimental results show that the proposed engine can handle look-ahead assertions efficiently.

V. Conclusion

We proposed a matching method for look-ahead assertions, which was combined with the existing systolic based matching engine [6]. In the proposed matching method, we introduce a preprocessing circuit to the systolic based engine. It performs matching for look-ahead assertions by searching from the end of a text to the beginning of the text. We also proposed a new buffer organization using stack memory which is suitable for the proposed matching method. The experimental results show that the proposed engine can handle look-ahead assertions efficiently.

References

[1] Sourcefire, Inc.,“SNORT Network Intrusion Detection System,” http://www.snort.org/.

[2] J. Bispo, I. Sourdis, J.M.P. Cardoso, S. Vassiliadis, “Regular expression matching for reconfigurable packet inspection,” Proc. 2006 IEEE ICFPT, pp.119-126, 2006.

[3] R. Sidhu, V.K. Prasanna, “Fast Regular Expression Matching Using FPGAs,” Proc. 2001 IEEE FCCM, pp.227-238, 2001.

[4] C.R. Clark, D.E. Schimmel, “Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns,” Proc. 2003 IEEE ICFPL, pp.956-959, 2003.

[5] Y.K. Chang, C.R. Chang, C.C. Su, “The Cost Effective Pre-processing Based NFA Pattern Matching Architecture for NIDS,” Proc. 2010 IEEE ICAINA, pp.385-391, 2010.

[6] Y. Wakaba, M. Inagi, S. Wakabayashi, S. Nagayama, “An Efficient Hardware Matching Engine for Regular Expression with Nested Kleene Operators,” Proc. 2011 IEEE ICFPL, pp.157-161, 2011.

[7] J. Divyasree, H. Rajashekar, Kuruvilla Varghese, “Dynamically reconfigurable regular expression matching architecture,” Proc. 2008 ASAP, pp.120-125, 2008.

[8] J. E. Hopcroft, R. Motwani, J. D. Ullman, “Introduction to Automata Theory, Languages, and Computation (3rd Edition),” Pearson Education, 2006.