Regular Expression Matching for Reconfigurable Constraint Repetition Inspection

Miad Faezipour and **Mehrdad Nourani**
Center for Integrated Circuits & Systems
The University of Texas at Dallas, Richardson, TX 75083
{mxf042000,nourani}@utdallas.edu

***Abstract*— Recent network intrusion detection systems (NIDS) use regular expressions to represent suspicious or malicious character sequences in packet payloads in a more efficient way. This paper introduces a new basic building block based on Non-deterministic Finite Automata (NFA) hardware implementation to support complex constraint repetitions in regular expressions. This block is a customized counter capable of handling any type of constraint repetition, applicable to any sub-regular expression. We also introduce optimization techniques to reduce the area and improve the overall performance. We have examined SNORT IDS regular expressions by taking advantage of the basic NFA building blocks, our proposed counting block and our proposed optimization techniques. We report experimental results for our architecture that verify area saving and performance improvement.**

***Index Terms*— Network Intrusion Detection System, Non-deterministic Finite Automata, Regular Expression.**

I. Introduction

A. Background

Regular expression is, technically, a defined grammar that uses standardized syntax conventions to specify patterns [1]. Unlike static patterns, a regular expression (RegExp) can specify complex patterns of character sequences, thus making it attractive for use in complex pattern searching [2]. UNIX utilities and programming languages such as PERL have regular expressions as their key powerful feature. Regular expressions are extensively used in networking applications, due to their powerful expressiveness. One recent application is their use in network intrusion detection and prevention systems (NIDS/NIPS) to represent strings or patterns corresponding to malicious data. Snort IDS [3] analyzes packet headers, and further inspects packet payloads for any hazardous content. Nowadays, many IDS handle their desired rules in the form of regular expressions. For example, SNORT IDS rule-set contains over 500 regular expressions and over 2,000 static patterns [3] [4]. Snort IDS follows the Perl Compatible Regular Expression (PCRE) syntax.

String matching is one of the most computationally intensive tasks for intrusion detection. Since software approaches cannot meet the time budget for high data rates, they are considered highly inefficient for high-speed networking. Hardware solutions such as FPGA implementations are of more interest, due to their high throughput and reconfigurability.

DFA-based approaches require huge amount of hardware resources, and thus suffer from state-explosion. On the other hand, with their compact hardware structure that allow multiple active states at a time, NFAs indeed provide an attractive solution [4] [5]. In this paper, we focus on NFA-based approaches for RegExp matching circuits in hardware.

Ever since Sidhu and Prasanna [6] proposed basic building blocks such as (un-constraint repetition) meta-characters in NFA to implement regular expressions on an FPGA, many others have continued this interesting field of research. Though many techniques were presented to complete or optimize the hardware implementation of RegExp meta-characters [7] [8], there is still room for much more improvement.

B. Main Contribution and Paper Organization

Our contribution to NFA-based regular expression matching circuits is the design and implementation of a customized counting block to efficiently handle constraint repetitions in IDS regular expressions. Constraint repetitions are extensively seen across practical rule sets such as the current Snort v2.7 IDS rules [3]. The conventional act of unrolling the circuit to successive repetitions is highly inefficient. Instead, a counting mechanism customized for this purpose can significantly improve the area cost and performance.

Our counting block is different in many ways from the previously introduced counting feature described in [4] [9]. The novelty of our proposed block is threefold. First, it offers a customized counter that can take care of all types of constraint repetitions, namely *Exactly*, *At Most*, *At Least* and *Between* blocks. All these types of constraint repetitions are implemented in one block, rather than having separate units for each, as introduced in [4]. Second, our counting block is capable of applying any type of constraint repetition to any type of sub-regular-expression. To the best of our knowledge, this feature has not been addressed in earlier approaches. In [4] [9] [10], the counting block could only be applied to a single character, which limits the counter application to single character counts in SNORT IDS rules. A large percentage of SNORT rules contains constraint repetitions for single characters. Our counting mechanism has the capability of dealing with group character counts, which also exist in IDS rules. Third, we propose a more cost-efficient circuitry for the *Alternate* ("|") meta-character that is applied to a number of single characters in a pattern rule. This also applies to a range of numbers or range of characters (e.g. $[0-9]$ or $[a-z]$, $[A-Z]$, or $[A-Za-z]$). This optimization technique is especially useful when single character alternates occur within patterns that also contain constraint repetitions.

The rest of this paper is organized as follows. In Section II, we briefly take a glance at prior work related to network IDS regular expression matching circuits in hardware. Our customized counting block for constraint repetitions is proposed in Section III, and the overall architecture is explained in detail. We elaborate on the overlapping feature of the counter design in the same section. We introduce our optimization techniques for reducing the area of the *Alternate* meta-character and the counter unit in Section IV. Experimental results are summarized in Section V. Finally, concluding remarks are in Section VI.

II. Prior Work

Many researchers have investigated the regular expression matching circuits in hardware. Floyd et al. were the first to implement non-deterministic automaton (NFA) based regular expression matching in hardware [11]. Then, Sidhu and Prasanna [6] proposed the basic building blocks in NFA to implement regular expressions in hardware. They used a character comparator for each and every character in the RegExp, which resulted in high hardware cost. The design was capable of processing one character (one byte) per clock cycle. Later, Clark et al. [8] proposed the character decoder instead of the character comparator, to save much of hardware and interconnecting area. The authors also exploited parallelism to process multiple bytes per clock, which significantly improved the throughput. Sharing and various optimization techniques were proposed in [7], which lead to significant are savings.

NFA building blocks for counting meta-characters have been addressed in [4] [9] [10]. These approaches, however, can only be applied to single character patterns, and require the traditional unrolling mechanism for constraint repetitions applied to a group of characters. Authors in [12] discuss the difficulties in applying the counting meta-character to a group of characters. They suggest a control unit to keep track of the number of characters (states) in the sub-pattern. This solution was eventually useful to detect constraint repetitions applied to only sub-patterns with finite lengths. The authors mentioned that the generic problem of implementing a matching circuit for constraint repetitions applied to a group of characters with unknown and infinite lengths, remains as an open issue. In this work, we focus on the counting meta-character and propose solutions to the problems others have encountered.

III. Counting Meta-character Design

The counting meta-character, basically, looks for successive matches of a specified sub-RegExp in any form of the four types of constraint repetitions. A brief description of all four types of constraint repetitions is provided in Table I where "(Sub-RegExp)" denotes the sub-regular expression that the constraint repetition is applied to. Our counter block is designed to handle all types of constraint repetitions that may appear in regular expressions. Similar to other NFA-based approaches that can process at least one character (one byte) per clock cycle, our design processes one character per cycle, thus yielding the same time complexity.

The conventional approach to deal with constraint repetitions is to unroll the pattern into the number of repetitions required. However, constraint repetitions of nearly one thousand repetitions or more have been seen across SNORT IDS rules [4]. This can easily consume a huge portion of hardware resources, and is clearly inefficient. In addition to the inefficient unrolling operation, some sort of controlling mechanism is inevitably required for the hardware implementation of the

TABLE I
Types of constraint repetitions in RegExp's.

Type of Constraint Repetition	Notation	Example
Exactly	(Sub-RegExp)$\{n\}$	(abc)$\{3\}$
At Most	(Sub-RegExp)$\{,n\}$	(b)$\{,100\}$
At Least	(Sub-RegExp)$\{n,\}$	$[\hat{}\backslash n]\{1000,\}$
Between	(Sub-RegExp)$\{n,m\}$	(abc)$\{3,5\}$

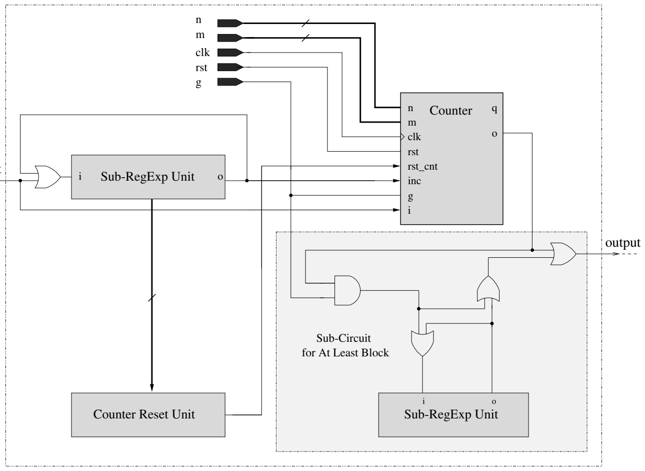


Figure 1. Architecture of counting meta-character.

At Most, *At Least* and *Between* constraint repetition types. This is why it is essential to design a counting block to handle constraint repetitions more efficiently.

Figure 1 illustrates the overall architecture of our counting block. The input to the design is the output of the previous sub-RegExp, which is *OR*ed with the output of the sub-RegExp that the constraint repetition is being applied to. This output signal is fed to a counter block which is incremented whenever the desired sub-RegExp has been detected. Signal q is the counting value of our counter block. Essentially, the counter block value q is incremented for successive matches of the sub-RegExp pattern. We now explain in detail all different units of our counter building block.

A. Sub-RegExp Unit

Sub-RegExp is the pattern string that the counting meta-character is being applied to. This can be any sub-string such as a single character, a group of characters, a sub-regular expression having fixed or variable length characters, or even strings containing meta-characters. In our design, whenever the sub-RegExp is detected, the *inc* signal becomes high, which in turn, increments the counter. See Figure 1.

B. Counter Reset Unit

This unit is directly related to the sub-RegExp pattern, and is configured when the sub-RegExp is defined. For a single character sub-RegExp, this unit is simply an inverter attached to the output of the sub-RegExp unit. However, for a group of characters, it is more than just an inverter. The circuit consists of multiple states that identifies a mismatch for the sub-RegExp. Basically, if the input stream contains any character other than the ones in the sub-RegExp, or includes the sub-RegExp characters, but messes the consecutive property that the sub-RegExp count demands, the counter should reset. As an example, consider the RegExp "$d(abc)\{n\}$", where n is an arbitrary number. Any incoming character other than a, b, or c should reset the counter. In addition, if any of a, b or c characters come but the order in which they appear differs from the *abc* sequence, the counter should reset. Moreover, when the counter has reached the predefined m value, it should also reset. The sub-circuit to reset the counter can be easily designed considering the above conditions. In Figure 2, the dashed box shows the logic circuit to reset the counter for RegExp "$d(abc)\{n\}$". Note that in this RegExp, "$(abc)\{n\}$" is preceded by character d. Therefore, the flip-flop and *AND* gate for character d is connected to the input of sub-RegExp "$(abc)\{n\}$", as shown in Figure 2. In order to generalize the counter reset circuit for any sub-RegExp, all we need is to generate the negation of the intermediate states to produce the non-consecutive property for the sub-RegExp count.

C. Counter

This is a customized counter unit that increments the count value q on the rising edge of the clock, if *inc* signal is active. This signal becomes high whenever the sub-RegExp is detected. The counter has a global reset signal (*rst*) as well as a local one (*rst_cnt*). The global reset is used for power-on initialization. The local reset signal resets the counter whenever the reset sub-circuit (explained in the previous subsection) becomes active. The counter is also designed such that if the counter has reached its maximum value m, the counter should reset through this signal. The counter takes n and m as inputs to determine the range of the count when needed. Controlling signals to the counter *inc* and *rst_cnt* are generated within other units of the design, as discussed earlier. Signal g is an input that indicates whether the constraint repetition is of the *At least* type or not. Signal o is the final output of the design, which indicates when the sub-RegExp containing the counting meta-character has been detected. Logic Equation for output signal o can be written as:

$$o = \begin{cases} 1 & \text{if} \quad (n \leq q \leq m) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

This Equation indicates that depending on the constraint repetition type, signal o remains high when the count value q is in between n and m.

D. Sub-Circuit for At least Block

The *At Least* block with notation "(sub-RegExp)$\{n,\}$" can be written as "(sub-RegExp)$\{n\}$(sub-RegExp)*". As we have the *Exactly* type implementation through the counter, for *At Least* block we use an *Exactly* block followed by zero or more repetitions of the sub-RegExp. Thus, in this unit, signal g is used to *OR* the counter output with the "(sub-RegExp)$\{n\}$(sub-RegExp)*" sub-circuit.

In summary, the parameters in our counting mechanism can be classified as follows:

- **Exactly** n**:** (sub-RegExp){n}, where $m = n$ and $g = 0$.
- **At Most** n**:** (sub-RegExp){,n}, where $n = 1$, $m > n$ and $g = 0$.
- **At Least** n**:** (sub-RegExp){n,}, where $m = n$ and $g = 1$.
- **Between** n **and** m**:** (sub-RegExp){n,m}, where $m > n$ and $g = 0$.

E. Dealing with Overlaps

Overlapping in RegExp patterns is defined as conditions where the input stream contains the RegExp pattern that itself, appears within the same RegExp pattern [13]. Detecting overlapping conditions is important for an IDS, since an attacker can execute the attacks that may be overlooked by the overlapping condition [13].

Our counting building block for the constraint repetition meta-character does not need to have the capability of detecting overlapping matches, and thus could save hardware resources cost. There are only three locations where a constraint repetition may be placed in a RegExp rule: the *beginning*, *in between*, or *at the end*. Having a constrained repetition of the *Exactly* or *Between* type at the *beginning* of a RegExp pattern may lead to a mismatch if overlaps exist. To overcome this, in the rare case where a RegExp should begin with the *Exactly* or *Between* types of constraint repetitions, the *At Least* block can be implemented instead, to avoid mismatches caused by overlapping conditions. When a constraint repetition of the *Exactly* or *Between* type occurs *in between* a RegExp string, overlaps would lead to wrong detection of the string. Thus, finding overlaps is not only useless for this case, but also misleading, producing a *false positive*. The case where constraint repetitions are in between a RegExp pattern rule is the majority case in SNORT IDS rules, which can be handled by our design. Overlaps located *at the end* of a pattern may not be useful either, and do not add any value to the RegExp detection. Thus, these type of overlaps are not of any concern either.

IV. Optimizations for Area Reduction

In this section we introduce two optimization techniques to improve the area cost of our counter design.

A. Alternate Meta-character

The *Alternate* (union) meta-character "|" is used to *OR* a group of sub-regular expressions. The conventional way of implementing this meta-character is the approach that Sidhu et al. [6] presented as one of their NFA basic building blocks. Figure 3 (a) shows the conventional implementation of the "$(a|b|c|d)$" RegExp. As an alternative, the character lines from the character decoder could be *OR*ed at the beginning, and the character flip-flops could be shared. Figure 3 (b) shows our optimized circuit for "$(a|b|c|d)$". Note that this optimization technique can only be applied to the alternation of single character patterns, including alpha-numeric ranges. Rules that require the negation of a character class (e.g. $[^\wedge 0-9]$ can also take advantage of this optimization technique by *AND*ing (instead of *OR*ing) the negated characters at the beginning. Alternation of groups of characters still requires the conventional implementation style, because unlike single characters, a group of characters cannot be directly taken from the character decoder. Thus, they cannot be *OR*ed at the beginning, and hence, our optimization technique cannot

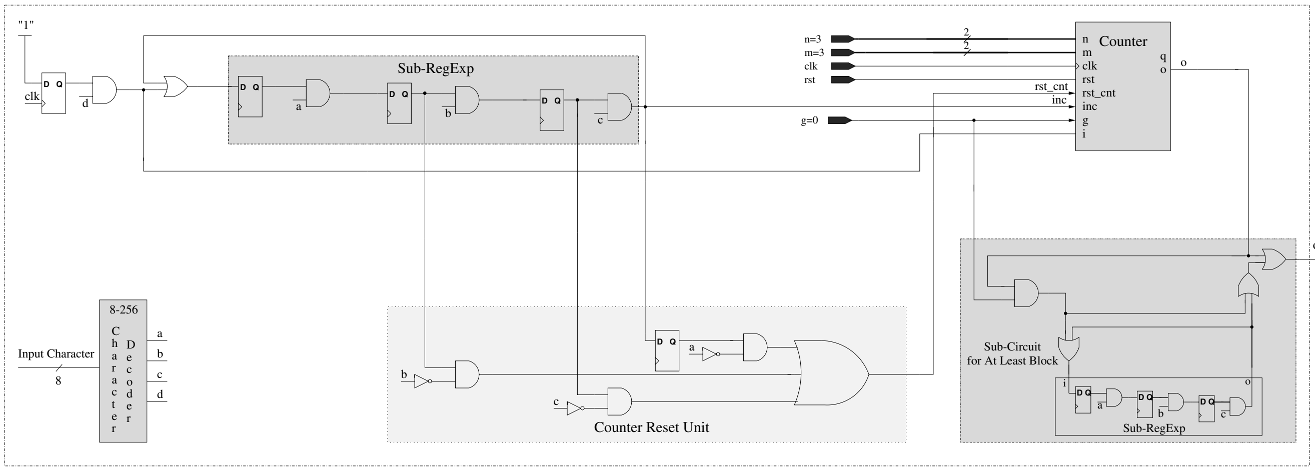
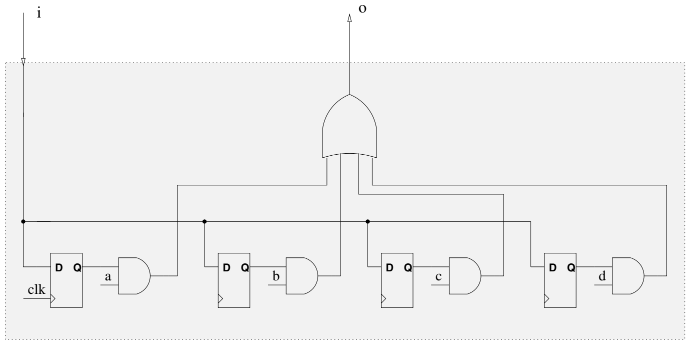
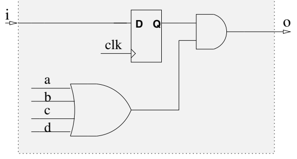


Figure 2. Sub-circuit configuration for counter reset for RegExp “$d(abc)\{3\}$”.





(a) Conventional. (b) Optimized.

Figure 3. Implementation of “$(a|b|c|d)$” RegExp.

be applied to them. Fortunately, a large portion of the SNORT IDS rules contain the alpha-numeric ranges as well as other single character alternations. In addition, many SNORT IDS RegExp rules have the single character alternated within constraint repetitions (see Section V). By taking advantage of this optimization technique, the area cost of these type of circuits can be significantly reduced compared to the conventional approaches.

B. At Least Block

We have previously implemented the *At Least* (sub-RegExp)$\{n,\}$ circuit by implementing the sub-RegExp *Exactly* block, followed by zero or more repetitions of the sub-RegExp circuit (see Section III-D). This requires an extra replica of the sub-RegExp unit plus a few gates, which is costly if used for every constraint repetition in RegExp’s. Note that more than 70% of the constraint repetitions in SNORT v2.7 IDS rules are of the *At Least* type [3]. Therefore, having an area efficient design for the Counter unit is essential. The Counter unit is designed such that output signal o remains high when the value of q is between n and m (based on what type of constraint repetition is desired). However, we can remove the sub-circuit that was required for the *At Least* block implementation by effectively taking signal g into account, directly in the Counter unit. Output signal o should be high for all type of constraint repetitions except the *At Least* type when q is between n and m, and should remain high when the q value has reached m or higher for the *At Least* type. Note that the count value q is incremented whenever successive repetitions of the sub-RegExp has occurred. To avoid overflow, we intentionally remain in count value $q = m$ after the count has reached m, for the *At least* type only. The *At Least* block resets the counter any time the local counter reset signal *rst_cnt* becomes active. The logic relationship [1] for output signal o can now be written as:

$$o = \begin{cases} 1 & \text{if} \quad (\bar{g} \cdot (n \leq q \leq m)) + (g \cdot (q = m)) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

This design is much more cost efficient compared to the earlier one, as it only adds a few logic gates while omitting the extra sub-circuit for *At Least* block entirely.

V. Experimental Results

Since the largest value of constraint repetitions noticed in SNORT IDS rules is 2082 (slightly greater than 2048) [4], a 12-bit counter would be required in the worst case. This meta-character unit would obviously count as a very small portion of the entire regular expression matching circuit. Our design can process one byte per clock cycle, hence the hardware implementation can easily achieve overall throughput of multiple *Gbps*, just like other NFA-based approaches.

[1]Symbols $\cdot$ and $+$ stand for Boolean notations of logic *AND* and logic *OR*, respectively.

TABLE II

AREA COMPARISON FOR THREE SNORT REGEXP'S USING OUR APPROACH AND THE CONVENTIONAL DESIGN.

RegExp	Approach	*Cost [Gates]*	*Area Saving*
RegExp (i)	Conventional	1918	-
	Ours	432	77.48%
RegExp (ii)	Conventional	2892	-
	Ours	597	79.35%
RegExp (iii)	Conventional	5540	-
	Ours	1004	81.94%

(i) `/\x28\s*name\s*\x22[^\x22]{260,}/smi`

(ii) `/^http\x3a\x2f\x2f[^\n]{400}/smi`

(iii) `/^Content-Disposition\x3a(\s*|\s*\r?\n\s+)[^\r\n]*\{[\da-fA-F]{8}(-[\da-fA-F]{4}){3}-[\da-fA-F]{12}\}/smi`

The area saving of our optimization technique for alternation is directly related to the number of single characters that are being alternated. It is clearly seen that if x single characters are alternated in a RegExp using the *Alternate* meta-character, the conventional circuit would contain x flip-flops and x *AND* gates, plus an x-bit *OR* gate and lots of interconnects. The optimized circuit would only contain one flip-flop and one *AND* gate (no matter how large the value of x is), plus the x-bit *OR* gate and very few interconnects. Hence, our area saving compared to the conventional approach is:

$$\Delta A = \frac{Total\ Unoptimized\ Logic - Total\ Optimized\ Logic}{Total\ Unoptimized\ Logic} \tag{3}$$

To measure the area saving, three different regular expressions from SNORT rules v2.7 [3] have been examined. We have used our counter mechanism and have taken our optimization techniques (optimized counter unit and optimized *Alternate* meta-character) into account using Equation 3. Table II compares the area cost of our approach versus the conventional approach in terms of 2-input NAND gate count. The last column in Table II clearly verifies our area reduction compared to the conventional design. The area savings increase for practical (e.g. SNORT) rules where the number of iterations of the constraint repetitions, or the number of single character alternations is quite large.

Overall, 52% of the entire SNORT IDS RegExp rule database contains counting meta-characters and single character alternates (or character classes and ranges). Table III shows the statistics on SNORT IDS v2.7 (as of Feb. 2008) [3] rules. Three common rule sets (oracle, web-misc and web-cgi) have been analyzed and the estimated area savings achieved by applying our mechanism are summarized in this table.

VI. CONCLUSION

We introduced an efficient counting block for constraint repetitions in regular expressions. Our optimized counting block can handle all four major types of constraint repetitions that appear in regular expressions. The counter block had the capability of being applied to any sub-RegExp and was not suffering from overlapping conditions, making it highly efficient for hardware implementation of SNORT IDS rules. Furthermore, an optimized circuit for the *Alternate* meta-character was presented, which can save a large amount of hardware resources when applied to single character patterns. Simulations results verify that our approach achieves up to 85% area savings for the same SNORT IDS RegExp rules implemented using the conventional approach, without performance degradation.

TABLE III

STATISTICS ON SNORT IDS RULES

Rule Set	*# of Patterns*	*# of Rules*	*# of Repetitions*	*Area Saving*
oracle	341	287	1,821	85.17%
web-misc	497	72	92	53.87%
web-cgi	456	12	4	55.11%

REFERENCES

[1] "Regular Expression Sample Application - User Search," *http://www.oracle.com/technology/sample_code/tech/pl_sql/regexp/usersearch/readme.html.*

[2] "An Introduction to Regular Expression with VBScript," *http://www.4guysfromrolla.com/webtech/090199-1.shtml.*

[3] SNORT Network Intrusion Detection System, *www.snort.org*.

[4] J. Bispo, I. Sourdis, J. M. P. Cardoso and S. Vassiliadis, "Regular Expression Matching for Reconfigurable Packet Inspection," in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT'06), pp. 119-126*, Dec. 2006.

[5] J. E. Hopcroft and J. D. Ullman, "Introduction to Automata Theory, Languages and Computation," *Reading, Mass.: 2nd Edition, Addison Wesley*, 2001.

[6] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), pp. 227-238*, Apr. 2001.

[7] C.-H. Lin, C.-T. Huang, C.-P. Jiang and S.-C. Chang, "Optimization of Pattern Matching Circuits for Regular Expression on FPGA," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 15, no. 12, pp. 1303-1310*, Dec. 2007.

[8] C. R. Clark and D. E. Schimmel, "Scalable Pattern Matching for High Speed Networks," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04), pp. 249-257*, Apr. 2004.

[9] I. Sourdis, S. Vassiliadis, J. Bispo and J. M. P. Cardoso, "Regular Expression Matching in Reconfigurable Hardware," in *Journal of VLSI Signal Processing, pp. 1-23*, July 2007.

[10] A. C. Mihal, C. Sauer and K. Keutzer, "Designing a Sub-RISC Multi-Gigabit Regular Expression Processor," *Technical Report, University of California at Berkeley, no. UCB/EECS-2006-119*, Sep. 2006.

[11] R. W. Floyd, and J. D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits," *Journal of ACM, vol. 29, no. 3, pp. 603-622*, July 1982.

[12] J. Bispo, I. Sourdis, J. M. P. Cardoso and S. Vassiliadis, "Synthesis of Regular Expressions Targeting FPGAs: Current Status and Open Issues," in *Proceedings of the 3rd International Workshop on Applied Reconfigurable Computing: Architectures, Tools and Applications (ARC'07), pp. 179-190*, March 2007.

[13] Z. K. Baker, V. K. Prasanna and H.-J. Jung, "Regular Expression Software Deceleration for Intrusion Detection Systems," in *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications (FPL'06), pp. 1-8*, Aug. 2006.