Constraint Repetition Inspection for Regular Expression on FPGA

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 implemented SNORT IDS regular expressions in hardware 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, Constraint Repetition Inspection.**

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 (patterns that are not expressed in the regular expression form) [3] [4]. Snort IDS follows the Perl Compatible Regular Expression (PCRE) syntax. Consider: `alert tcp $HOME_NET 5400 -> $EXTERNAL_NET any; pcre:"/^Blade\s+Runner\s+ver\s+\d+/smi"` which is a v2.7 SNORT IDS rule [3]. This rule warns of any packet payload content that includes a string matching regular expression `"/^Blade\s+Runner\s+ver\s+\d+/smi"`. Notations such as ^ and + stand for specific character meanings in RegExp's. See Table I for more explanation. In this particular example, the RegExp matches patterns that begin with `Blade`, followed by one or more whitespace characters, followed by `Runner`, followed by one or more whitespace characters, followed by `ver`, followed by one or more whitespace characters, followed by one or more 0-9 digit characters, and then followed by `/smi`.

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.

Unlike Deterministic Finite Automata (DFA) based solutions that allow only one active state at a time, Non-Deterministic Finite Automata (NFA) designs allow multiple active states at a certain time. DFA-based approaches are attractive for sequential designs in mostly software solutions, which require only one active state at a time. On the other hand, NFA-based approaches well suit parallel architectures, due their inherent structure of allowing multiple active states at a time. This feature makes NFA solutions highly suitable for hardware designs [4]. Moreover, if the DFA set of input symbols (which is 2^8 symbols when considering the extended ASCII codes) is expressed as Σ, a DFA would require up to $O(\Sigma^n)$ states to represent a regular expression of length n in the worst case [4] [5]. The same RegExp would only require $O(n)$ states in a NFA representation [5], which is a huge advantage. Briefly, DFA-based approaches require huge amount of hardware resources, and thus suffer from state-explosion. On the other hand, with their compact hardware structure, NFAs indeed provide an attractive solution. In this paper, we focus on NFA-based approaches for RegExp matching circuits in hardware.

B. Main Contribution and Paper Organization

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] [9], there is still room for much more improvement. Our contribution to this matter 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.

1550-4794/08 $25.00 © 2008 IEEE
DOI 10.1109/HOTI.2008.14

TABLE I
LIST OF BASIC META-CHARACTERS/SYNTAX IN REGULAR EXPRESSIONS.

Meta-Character/Syntax	Definition
Kleene Star "$*$"	Zero or more repetitions of the preceding Sub-RegExp [6].
Concatenation	A Sub-RegExp followed by another Sub-RegExp [6].
Alternation "|"	Union (*OR*) of two or more Sub-RegExp's [6].
Question Mark "?"	Zero or one repetition of the preceding Sub-RegExp [13].
Plus "+"	One or more repetitions of the preceding Sub-RegExp [7].
Dot "."	Matching any character except newline [4].
Negation "^"	All characters except the following Sub-RegExp in square brackets [7].
Start "^"	Matching the following Sub-RegExp at the beginning of a string after newline [4].
Dollar "$"	Matching the end of a pattern stream followed by linefeed and carriage return [7].
Backslash "\"	Escapes the following meta-character, returning to its literal meaning [4].
Count	Constraint repetition of the preceding Sub-RegExp (partially implemented in [4] [10] [11]).
$\backslash s$	Matching the whitespace character [4] [6].
$\backslash d$	Matching any of the 0-9 digit characters [4] [6].
$\backslash w$	Matching any word character including letters and digits [4] [6].
$\backslash n$	Matching the newline (linefeed) character [4] [6].
$\backslash r$	Matching the carriage return character [4] [6].
$\backslash t$	Matching the tab character [4] [6].
$\backslash x$	Matching the hexadecimal value that follows [6].

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] [10]. 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] [10] [11], 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 [12]. 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. [9] 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.

Optimization techniques such as sharing common substrings of RegExp rules has been extensively studied in [13] [7]. In [13], the authors take advantage of sharing prefix patterns, and achieved an area reduction of 37% compared to the conventional approach [6]. The authors used JHDL (a JAVA-Based Design Tool) to describe circuits by writing a JAVA code that constructs the circuit via JHDL libraries. Authors in [7] [8] introduced controlling units to efficiently take advantage of sharing infix patterns, as well as sharing common prefix patterns. These techniques led to an area reduction of 70% compared to the conventional approach.

In contrast to NFA-based approaches, DFA-based solutions have also been widely used to design RegExp matching circuits. Moscola *et al.* [14] developed a content scanning module using DFAs to implement static patterns for IDS. The authors used the JHDL tool to construct the hardware more efficiently. Authors in [15] designed a custom microcontroller to implement regular expression matching circuits in hardware. Their approach was a DFA-based approach that stored patterns in memory tables, which gave the reconfigurable capability to update regular expressions at run-time. Authors in [16]

[17] proposed minimization and pattern re-write techniques to reduce state explosion conditions that may occur in DFA-based approaches.

Customized logic circuits and Content Addressable Memories (CAM) has been another solution to efficient pattern matching in hardware. Authors in [18] [19], use CAMs and Ternary CAMs, respectively, to achieve giga-bit rate pattern matching engines highly efficient for network security. However, these approaches allow static pattern matching, and are not very feasible for RegExp matching. The ternary feature of TCAMs that has the capability of storing *don't-cares* in addition to 1's and 0's, do not allow much of flexibility as desired for RegExp patterns. Limited regular expression matching is provided using these techniques. In this regard, Sourdis *et al.* [20] proposed a hybrid approach to initially decode patterns before the CAM-based search, and used pipelining techniques, to achieve higher frequency throughputs.

The main RegExp meta-characters and syntax notations are listed in Table I. Definition and reference to the papers that proposed the NFA building block for each meta-character/syntax has also been provided in this Table. Note that the meta-character implementations should be applied to any sub-regular expression directly. The counting meta-character has been addressed in [4] [10] [11]. 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 [21] 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 II where "(Sub-RegExp)" denotes the sub-regular expression that the constraint repetition is applied to. For example, the last row in Table II means finding a match for the range of 3 to 5 successive repetitions of the substring "*abc*". Our counter block is designed to handle all types of constraint repetitions that may appear in regular expressions.

The conventional approach to deal with constraint repetitions is to unroll (or cascade) 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

TABLE II
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)$\{3,5\}$

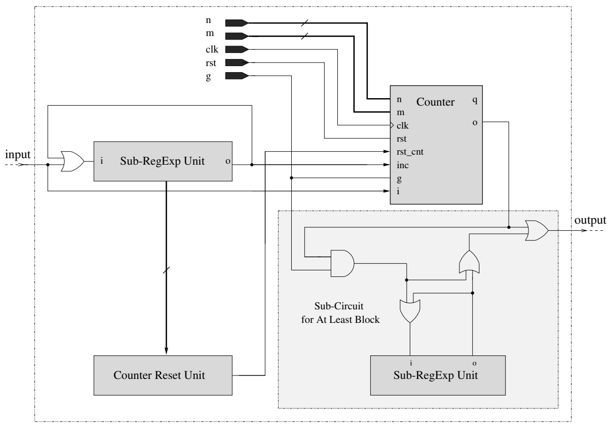


Figure 1. Architecture of counting meta-character.

mechanism is inevitably required for the hardware implementation of the *At Most*, *At Least* and *Between* constraint repetition types. For example, consider the *Between* type RegExp "$(abc)\{3,5\}$". The conventional approach would be to consecutively replicate the matching circuit for concatenated characters "*abc*" five times. Moreover, a control circuitry (e.g. some *OR* gates) is also required to distinguish the third, fourth and fifth repetition of "*abc*". 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.

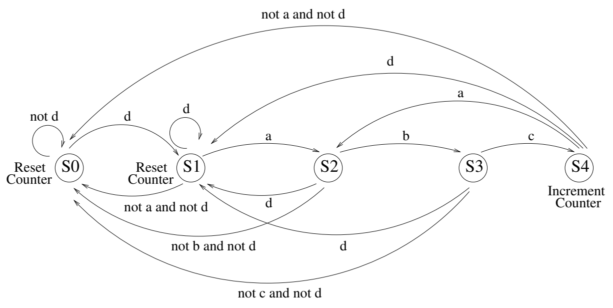


Figure 2. State diagram of the counting mechanism for RegExp "$d(abc)\{n\}$".

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. Figure 2 shows the state diagram for NFA construction of the RegExp for the counting mechanism. At states S0 and S1, the counter should reset. State S4 is when the counter should be incremented. The sub-circuit to reset the counter can be easily designed using this state diagram. In Figure 3, 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 3. 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 sub-section) 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:

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

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 [15]. For instance, the RegExp "*telephone* | *phonebook*" causes an overlapping condition for the input stream "*telephonebook*". The common term "*phone*" appears in both alternated patterns, and thus should result in two matches, one after phone, and the other after book. Detecting overlapping conditions is important for an IDS, since an attacker can execute the attacks that may be overlooked by the overlapping condition [15].

Unlike most DFA-based approaches, RegExp matching circuits using NFA basic building blocks inherently have the capability of detecting overlapping matches. This is due to the fact that in this approach, each character is processed per clock cycle through the character decoder and the designated character flip-flop. Thus, overlapping conditions in the input stream eventually get through by activating the character flip-flop multiple times, without missing any matching character. DFA-based approaches, however, would need more edges and possibly more states to take care of the overlaps.

We show that 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. To justify this, note that there are only three locations where a constraint repetition may be placed in a RegExp rule. The beginning, in between, or at the end. To be more clear, we consider these three cases and explain why overlaps are not of any concern:

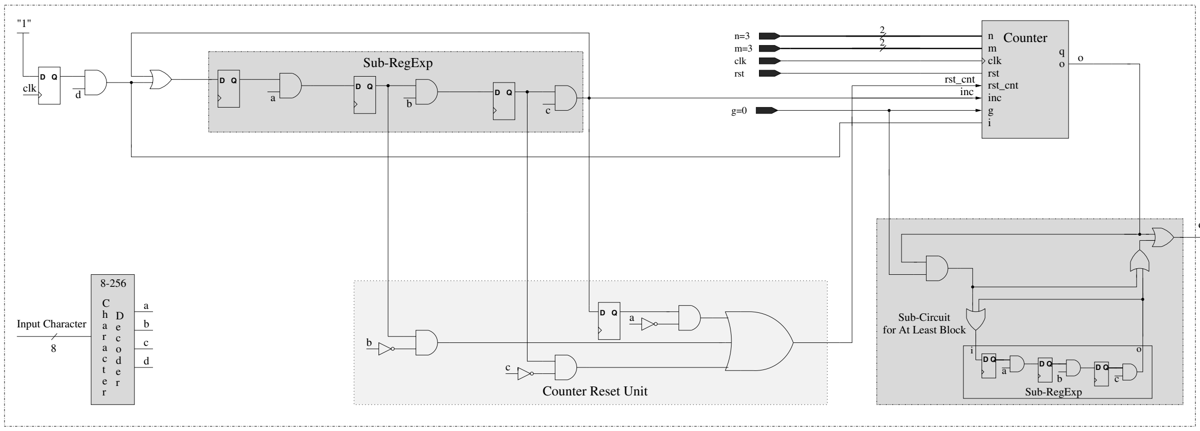


Figure 3. Sub-circuit configuration for counter reset for RegExp "$d(abc)\{3\}$".

- **At the Beginning:** Having a constrained repetition of the *Exactly* or *Between* type at the beginning of a RegExp pattern is indeed an issue, and may cause missing the detection of the string if overlaps exist. For example, consider the RegExp pattern: "$[\wedge\backslash n]\{100\}ba+$". If the input stream contains more than a hundred (e.g. 104) characters that are not the newline character ($\backslash n$), followed by b, and then followed by a number of a's, a match should be detected. Overlapping takes place here, and the counter should consider the last hundred characters that were other than newline. However, our counter would reset after the first hundred non-newline characters, and would start counting up to only four by the time the next characters (b followed by a number of a's) are received. This would lead to a mismatch. Practically, almost no rule in SNORT IDS database starts with an *Exactly* or *Between* type of constraint repetition, and hence, our counting block with no overlapping capability will not produce any match problems. Also, note that patterns starting with the *At Least* block will be able to detect overlapping conditions. This is due to the inherent structure of the *At Least* block that includes the Kleene -star character "*", which is not a constraint repetition factor, and is capable of detecting overlaps. In rare cases 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.

- **In Between:** 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. Consider the RegExp "$ab\{3\}c$", where a constraint repetition is located in the middle of the string. The detector should not report a match for the input stream "*abbbbc*", while if the counter had the overlapping detecting capability, it would have reported a match. 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.

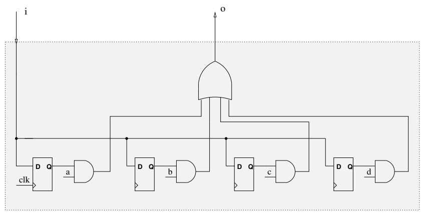
- **At the End:** When a constraint repetition is located at the end of a RegExp pattern, finding overlaps does not make any sense either. Authors in [10] also pointed out that overlaps located at the end of a pattern may not be useful. Let us consider the RegExp "$abc\{4\}$", where a constraint repetition of the *Exactly* type occurred at the end of the string. If the input stream is "*abccccc*", we should only get one match when the first four c's are detected right after ab, and not any longer. This is similar to the case where a constraint repetition is the only substring of a RegExp rule (which never occurs in SNORT IDS rules). In this case, we should note that overlaps don't really even matter. The reason is that if a match corresponding to a particular rule is found for an input stream, overlaps would just add to the number of matches. Reporting only one match for a rule is enough to mark the input stream suspicious. There is no reason to report that a stream is suspicious several times. Even though when considering overlaps, the locations of the matches may differ. They will still be very close to the first one found, since they were generated due to an overlapping condition that would be have been near the first one. A processing unit hereafter, takes care of analyzing the suspicious streams, and locates where the matches were found.

IV. Optimizations for Area Reduction

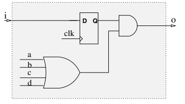
In this section we introduce two optimization techniques to improve the area cost of our counter design.

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 4 (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 4 (b)



(a) Conventional.



(b) Optimized.

Figure 4. Implementation of "$(a|b|c|d)$" RegExp.

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 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:

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

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.

TABLE III
ALTERA STRATIX II EP2S60F672C5 DEVICE UTILIZATION FOR A 12-BIT COUNTER BLOCK.

Resources	*Used*	*Available*	*Utilization Percentage*
Total ALUTs	56	48,352	<1%
Total Registers	14	51,182	<1%
Total Pins	43	493	9%

V. EXPERIMENTAL RESULTS

A. *Simulation of Counter Unit*

A 4-bit special counter unit was designed using Synopsys tools [22] to implement the RegExp "$d(abc)\{3,5\}$". Timing simulation in QuartusII environment [23] for input stream "*cadabadaabccadabcabcabcabcabcabcabcabccccc...*" [2] is shown in Figure 5. The waveform clearly shows the counting value q, and output o on each clock cycle. The constraint repetition in this RegExp is of the *Between* type, and thus, the output is high for counts 3, 4 and 5 of the sub-RegExp "*abc*".

Since the largest value of constraint repetitions inspected in SNORT IDS rules is 2082 (slightly greater than 2048) [4], a 12-bit counter would be sufficient in the worst case. Table III summarizes the hardware resource utilization of a 12-bit counter block using EP2S60F672C5 device of the Stratix 2S60 FPGA family series. As can be seen, our counter block is constructed using a very small portion of the logic cells available in the FPGA.

Based on the results reported by the tool, our system has a maximum clock frequency of $f_{max} = 368.32 MHz$. Since our design can process one byte per clock cycle, the system can achieve an overall throughput of $2.95 Gbps$.

B. *Area Savings*

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:

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

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

[2]The ASCII codes for a, b, c and d in hexadecimal are 61, 62, 63 and 64, respectively.

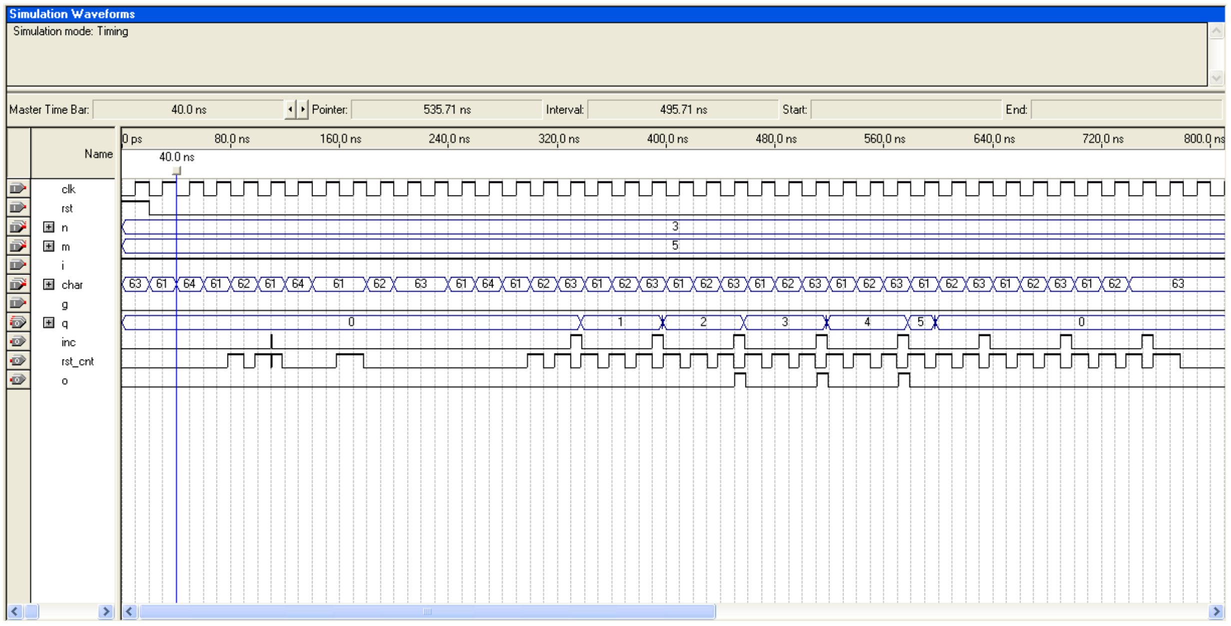


Figure 5. Timing simulations of the "$d(abc)\{3,5\}$" RegExp circuit for different input characters.

In our experimentation, the unoptimized case is conventional implementation of RegExp matching circuit [6]. The optimized scenario corresponds to our implementation in which the single character alternation and optimized *At Least* block in the counter are incorporated. If A_{ff} represents the area of a flip-flop, A_{AND} denotes the area of a 2-input *AND* gate, and the area of an x-bit *OR* gate is shown as $A_{OR_{x-bit}}$, our single character alternate area saving equation in percentage can be formulated as follows:

$$\Delta A = \frac{(x-1)\cdot A_{ff} + (x-1)\cdot A_{AND}}{x\cdot A_{ff} + x\cdot A_{AND} + A_{OR_{x-bit}}} \times 100\% \tag{4}$$

The sub-RegExp "$[A-Za-z]$" as one of the frequently used sub-patterns in SNORT IDS rules, alternates 52 single characters. Our optimization technique in this case results in 98% less flip-flops and 98% less *AND* gates compared to the conventional approach.

Timing characteristics of the optimized technique is similar to the un-optimized approach, and is not adversely affected. In the conventional approach, only one of the character flip-flops followed by an *AND* gate would activate the *OR* gate. Hence, the critical path of the optimized circuit (which goes through an x-bit *OR* gate, a flip-flop and an *AND* gate), remains the same as the conventional approach.

To measure the area saving, three different regular expressions from SNORT rules v2.7 [3] have been implemented on Altera FPGA. We have used our counter mechanism and have taken our optimization techniques into account. Table IV compares the area cost of our approach versus the conventional approach in terms of 2-input NAND gate count. The last column in Table IV 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.

TABLE IV
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`

Overall, 52% of the entire SNORT IDS RegExp rule database contains counting meta-characters and single character alternates (or character classes and ranges). Table V shows the statistics on SNORT IDS v2.7 (as of Feb. 2008) [3] and area savings achieved by applying our mechanism. Three common rule sets (oracle, web-misc and web-cgi), and also the entire SNORT IDS RegExp rule set, made of 53 sets, have been analyzed. Our approach leads to about 40% total area savings compared to the conventional (cascaded topology as explained earlier) approach.

VI. CONCLUSION

We introduced an efficient counting block for constraint repetitions in regular expressions. Our optimized counting

TABLE V
STATISTICS ON SNORT IDS RULES

Rule Set	*# of Static Patterns*	*# of RegExp Rules*	*# of Constraint Repetitions*	*Area Saving*
oracle	341	287	1,821	85.17%
web-misc	497	72	92	53.87%
web-cgi	456	12	4	55.11%
Entire SNORT (53 Rule Sets)	50,621	20,154	19,194	39.43%

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 some of SNORT IDS RegExp rules compared to the conventional approaches. Our approach leads to 40% area savings for the entire SNORT IDS rule set, without degrading the performance.

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 Regular Expression Pattern Matching Circuits on FPGA," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06), pp. 12-17*, Mar. 2006.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] B.L. Hutchings, R. Franklin and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02), pp. 111-120*, Sep. 2002.

[14] J. Moscola, J. Lockwood, R. P. Loui and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03), pp. 31-38*, Apr. 2003.

[15] 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.

[16] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley and J. Turner, "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection," in *Proceedings of the ACM Special Interest Group on Data Communication Conference (SIGCOMM'06), pp. 339-350*, Sep. 2006.

[17] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman and R. H. Katz, "Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection," in *Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS'06), pp. 93-102*, Dec. 2006.

[18] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," in *Proceedings of 12th International Conference on Field Programmable Logic and Applications (FPL'02), pp. 47-61*, Sep. 2002.

[19] F. Yu, R. H. Katz and T. V. Lakshman, "Gigabit Rate Packet Pattern-Matching Using TCAM," in *Proceedings of the 12th IEEE International Conference on Network Protocols, pp. 174-183*, Oct. 2004.

[20] I. Sourdis and D. Pnevmatikatos, "Pre-Decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," in *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04), pp. 258-267*, Apr. 2004.

[21] 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.

[22] Synopsys Inc., "User Manuals for SYNOPSYS Toolset Version 2005.06," 2005.

[23] ALTERA Corp., "User Manuals for Quartus II Version 6.0 Toolset", 2006.