Regular Expression Matching for Reconfigurable Packet Inspection

João Bispo [♮], Ioannis Sourdis [#], João M.P. Cardoso [*] and Stamatis Vassiliadis [#]

[#] *Computer Engineering, TU Delft,*
The Netherlands,
{sourdis, stamatis}@ce.et.tudelft.nl

[♮] *Faculty of Sciences and Technology, University of Algarve*
Faro, Portugal
joaobispo@gmail.com

[*] *Department of Informatics Engineering, INESC-ID/IST,*
Lisboa, Portugal
jmpc@acm.org

***Abstract*— Recent intrusion detection systems (IDS) use regular expressions instead of static patterns as a more efficient way to represent hazardous packet payload contents. This paper focuses on regular expressions pattern matching engines implemented in reconfigurable hardware. We present a Nondeterministic Finite Automata (NFA) based implementation, which takes advantage of new basic building blocks to support more complex regular expressions than the previous approaches. Our methodology is supported by a tool that automatically generates the circuitry for the given regular expressions, outputting VHDL representations ready for logic synthesis. Furthermore, we include techniques to reduce the area cost of our designs and maximize performance when targeting FPGAs. Experimental results show that our tool is able to generate a regular expression engine to match more than 500 IDS regular expressions (from the Snort ruleset) using only 25K logic cells and achieving 2 Gbps throughput on a Virtex2 and 2.9 on a Virtex4 device. Concerning the throughput per area required per matching non-Meta character, our design is 3.4 and 10× more efficient than previous ASIC and FPGA approaches, respectively.**

I. Introduction

High speed and always-on network access is becoming commonplace around the world, creating a demand for increased network security. Intrusion Detection Systems (IDS) such as Snort [1] are currently the most efficient solution for network security. Instead of only checking the header of each incoming packet, IDS also scan the payload of the packets to detect suspicious contents. These systems must be able to frequently update their ruleset with new descriptions of known attacks. This required flexibility and the fast processing rates needed might be only achieved using reconfigurable technology by exploiting specialized circuitry and parallelism.

In the past years, many researchers have worked on reconfigurable IDS focusing mostly on the payload scan, which turns out to be the most computationally intensive task [2]. Numerous techniques for reconfigurable IDS static pattern matching have been proposed [3]–[9]. Many of them employ regular expressions to represent the search static patterns, implementing either nondeterministic or deterministic finite automata (NFAs or DFAs) [4]–[6]. However, the representation of a single static pattern using a regular expression usually requires relatively simple regular expressions (RegExp) syntax such as concatenation, union operators $(a|b)$ or wildcards $(a*)$.

Although in the past years IDS used mostly static patterns to scan packet payloads, recently, regular expressions have become frequently utilized to describe more efficiently hazardous contents. For example, the recent ruleset of Snort IDS includes about 2,000 static patterns and more than 500 regular expressions which require complex syntax support such as *constrained repetitions* (i.e. $a\{10\}$, $a\{10+\}$, $a\{10, 12\}$). Although wildcards, Union and Concatenation operators have been efficiently implemented previously in hardware [3], the constrained repetitions are more complicated since they require keeping track of multiple states. One solution to this problem is the use of DFAs which allow only one active state at any time. However, a theoretical worst case study shows that a single regular expression of length n can be expressed as a DFA of up to $O(\sum^n)$ states (where $\sum$ is the DFA finite set of input symbols, that is 2^8 symbols for the extended ASCII code), while an NFA representation would require only $O(n)$ states [10]. Therefore, DFAs can produce inefficient designs in terms of area (logic or memory). On the other hand, when designed properly, NFAs can be more compact and area efficient. DFAs are more suitable for software solutions, since running sequential code makes difficult and slow to keep track of multiple active states. Since hardware is inherently concurrent, NFAs (implemented using specialized circuitries) is an attractive solution. This solution may fully exploit concurrency and it is relatively easy to keep track of multiple active states.

Our work addresses efficient implementations of NFA-based regular expression pattern matching engines. The main contributions of this paper are the following:

This work was partially supported by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6).

- We introduce new basic building blocks for *constrained repetition* operators, which are able to detect all overlapping matches. When combined with previous research in hardware NFA implementations, efficient designs can be achieved.
- To achieve more efficient hardware designs, we recompile the extracted IDS regular expressions and discard or replace all the syntax features that have been inserted to only accelerate software implementations (i.e. conditional branches, lookahead statements etc.).
- We employ several techniques to reduce the area requirements of our designs, such as RegExp prefix sharing, pre-decoding, centralized substring matching and character classes blocks, etc. We take advantage of Xilinx SRL16 shift registers to store multiple states with fewer hardware resources.
- We introduce a methodology to automatically generate the regular expression pattern matching engine from the regular expressions. We show how an hierarchical representation of the regular expressions is currently used to facilitate the automatic VHDL generation using basic building blocks. A tool that outputs the VHDL representation of the engine suitable for logic synthesis is presented.
- It shows we are able to generate efficient regular expression engines, in terms of area and performance, outperforming previous ASIC and reconfigurable hardware related approaches.
- Finally, we present a complete implementation of a payload scanner [1] for the SNORT v2.4 rule set (consisting of 2,200 static patterns and 509 RegExp).

The remainder of the paper is organized as follows: In section II, we discuss previous work on hardware regular expression matching and in section III we present the Snort-PCRE regular expression syntax. In Section IV, we describe the top-level approach of our regular expression engine, the basic building blocks and the techniques employed to reduce area and increase performance. In Section V, we present implementation results and compare them with related work. Finally, in Section VI we conclude the paper.

TABLE I
SNORT-PCRE BASIC SYNTAX CURRENTLY SUPPORTED BY OUR APPROACH.

Feature	Description
a	All ASCII characters, excluding meta-characters, match a single instance of themselves
[\^$.—?*+()	Meta-characters. Each one has a special meaning
.	Matches any character except new line
\?	Backslash escapes meta-characters, returning them to their literal meaning
[abc]	Character class. Matches one character inside the brackets. In this case, equivalent to (a|b|c)
[a-fA-F0-9]	Character class with range.
[^abc]	Negated character class. Matches every character except each non-Meta character inside breackets.
RegExp*	Kleene Star. Matches zero or more times the regular expression.
RegExp+	Plus. Matches one or more times the regular expression.
RegExp?	Question. Matches zero or one times the regular expression.
RegExp{N}	Exactly. Matches N times the regular expression.
RegExp{N, }	AtLeast. Matches N times or more the regular expression.
RegExp{N,M}	Between. Matches between N and M times the regular expression.
\xFF	Matches the ASCII character with the numerical value indicated by the hexadecimal number FF.
\000	Matches the ASCII character with the numerical value indicated by the octal number 000.
\d, \w and \s	PCRE Shorthand character classes matching digits 0-9, word characters (letters and digits) and whitespace, respectively.
\n, \r and \t	Match an LF character, CR character and a tab character respectively.
(RegExp)	Groups regular expressions, so operators can be applied.
RegExp1RegExp2	Concatenation. Regular Expression 1, followed by Regular Expression 2
RegExp1 | RegExp2	Union. Regular Expression 1 or Regular Expression 2.
^RegExp	Matches Regular Expression 1 only if at the beginning of the string.
RegExp$	Dollar. Matches Regular Expression only if at the end of the string.
(?=RegExp), (?!RegExp), (?<=text), (?<!text)	Lookaround. Without consuming characters, stops the matching if the RegExp inside does not match.
(?(?=RegExp) then |else)	Conditional. If the lookahead succeeds, continues the matching with the then RegExp. If not, with the else RegExp.
\1, \2. . .\N	Backreferences. Have the same value as the text matched by the corresponding pair of capturing parethesis, from 1st through Nth.
Flags	**Description**
i	Regular Expression becomes case insensitive.
s	Dot matches all characters, including newline.
m	^ and $ match after and before newlines.

II. RELATED WORK

Matching Regular Expressions in hardware has been widely studied in the past. In 1982, Floyd and Ullman discussed the implementation of NFAs in hardware [11], proposing among other aspects an hierarchical implementation produced by the McNaughton-Yamada algorithm [12]. More recently, Sidhu and Prasanna presented an FPGA implementation to match regular expressions formulated in NFAs and designed the basic blocks for Concatenation, Kleene-star and Union operators. Franklin $et\ al.$ used NFAs to represent all the Snort static patterns into a single regular expression, requiring substantially lower area [4]. Moscola $et\ al.$ used DFAs to match static patterns, since they discovered that static patterns can be represented in DFAs of practically $O(n)$ states [5]. Clark and Schimmel used pre-decoding to share the character comparators of their NFA implementations and thus reducing even more hardware resources [6]. Lin $et\ al.$ minimized the area cost of their NFA designs by sharing parts of the regular expressions [13]. Brodie $et\ al.$ converted the IDS patterns and RegExp into DFAs and implemented them with pipelined FSM structures specially designed for regular expression matching [14]. Their architecture uses memories to store transition and indirection tables and aims an ASIC implementation. Finally, Baker $et\ al.$ described a microcontroller implementation in FPGA for matching IDS regular expressions using DFAs [15]. Their design updates its ruleset by only changing the memory contents. The Snort RegExp are converted to DFAs in order

[1] assuming an extracted stream of packet payloads, after reassembling and reordering of packets.

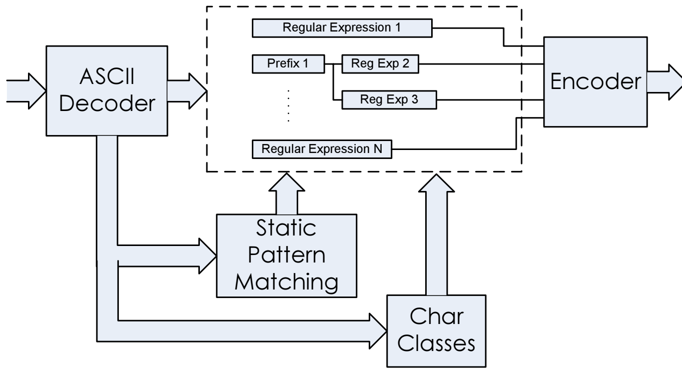

Fig. 1. Block diagram of our Regular Expression Engine.

to be ported into the proposed microcontroller. Some of the related works use DFAs to match the IDS patterns resulting in designs with significant area/memory requirements [5], [14], [15]. The rest employ NFAs without solving the problem of constrained repetitions and they consequently repeat the same circuit to implement the above syntax [16].

III. Snort Perl-compatible Regular Expressions

In this section, we discuss the regular expressions used in packet payload scanning. More precisely, we describe the features of the regular expressions included in Snort IDS. Snort adopted the Perl-compatible regular expression syntax (PCRE) [17]. For example, `alert tcp any -> (pcre:"/^PASS\s*\n/smi";)` is a Snort rule. Based on the above rule, Snort will detect any packet with payload including a string that matches the "`/^PASS\s*\n/smi`" regular expression. Apart from the well known features of a strict definition of regular expressions, PCRE includes more features such as constrained repetitions and several flags. Table I describes the PCRE basic syntax supported by our regular expression pattern matching engines. The PCRE syntax not currently supported is related to some anchors (\A, \Z, \z), word boundaries (\b, \B), differences between Greedy and Lazy quantifiers (we report both matches), continuing from the previous match (\G), and some Snort specific flags (\U, \R and \B). Since current Snort ruleset does not use these features, not currently supported by our engines, we have been able to implement a regular expression matching engine including all the 509 regular expressions of Snort. As described in the following section, the flags determine the functionality of some blocks, while the constrained repetitions and the rest of the features are either mapped directly to hardwired blocks or replaced with equivalent descriptions that suit our hardware implementation.

IV. Regular Expressions Engine

In this section, we describe our regular expression engine. Figure 1 depicts the top-level diagram of our design. The incoming data feed a centralized ASCII decoder (8 to 256 bits). The output of the decoder provides a single wire per character to the regular expression modules. This way, we match each character only once and all the regular expression modules receive the output lines from the decoder. For each Snort regular expression there is a separate RegExp module. Regular expressions with common prefixes share the same prefix sub-module. The static strings (more than one character) that are included in the regular expressions are matched separately in a DCAM (Decoded CAM) static pattern matching module described in our previous work [7]. DCAM pre-decodes incoming characters, aligns (shifts) the decoded data and ANDs them to produce the match signal for each pattern. Resource sharing is due to the centralized ASCII decoder and the shared shift registers. We match the sub-patterns using DCAM instead of PHmem [8] (Perfect-Hashing memory which is a more area efficient technique) because it can be integrated easier with the rest of the Regular Expression Engine. Similarly, the character classes (union of several characters e.g. $(a|b)$) are also implemented separately and share their results among the RegExp modules. Both static pattern matching and character classes modules are feed from the ASCII decoder. Each RegExp module outputs a match for the corresponding regular expression and subsequently, all the partial matches are encoded on a priority encoder.

A. Basic NFA blocks

We describe next the basic building blocks of our approach. We use the blocks described by Sidhu and Prasanna for

TABLE II
The basic building blocks of our Regular Expression Engine.

Block	Description
Character	Matches a single character, based on the design of single character described in [3].
Dollar ($)	Validates the match if in the end of the packet/string. Based on the Character Block [3].
Dot	Matches any character except new line. Based on the Character Block [3] the input character is the "$newline$" ($\backslash n$) character inverted.
Caret (^)	Starts a match every time a packet/string arrives. Based on the Character Block [3], the input character is the "$beginning\ of\ packet$" character.
Character Class	Matches a set of characters. Based on the Character Block [3], the input character is one of the outputs of $character\ class$ module. The $character\ class$ module ORs the characters included in a character class.
RegexBlock	Encapsulates hardware blocks that implement regular expressions or sub-blocks of regular expressions.
Question (?)	One or zero times the regular expression, based on the design of Kleene-star ($r*$) described in [3]. The incoming OR gate (to the flip-flop) has to be removed, consequently, the input token (i) goes directly to the flip-flop.
Plus (+)	One or more times the regular expression, based on the design of Kleene-star ($r*$) described in [3]. The outgoing OR gate has to be removed, consequently, the output token (o) is the output of the flip-flop, instead of the output of the second OR gate.
Kleene (*)	Zero or more times the regular expression, as described in [3].
Exactly	Matches exactly N times. Constraint Repetition for single characters and sets of characters. Described in Section IV-A.
AtLeast	Matches atleast N times. Constraint Repetition for single characters and sets of characters. Described in Section IV-A.
Between	Matches between N and M times. Constraint Repetition for single characters and sets of characters. Described in Section IV-A.
OrBlock	Union operator for regular expressions, as described in [3].
Pattern	Matches a string of characters. It has an interface for the DCAM Module. The input token has to be delayed for N cycles through an SRL16 in order to be correctly aligned with the output of the static pattern matching module.

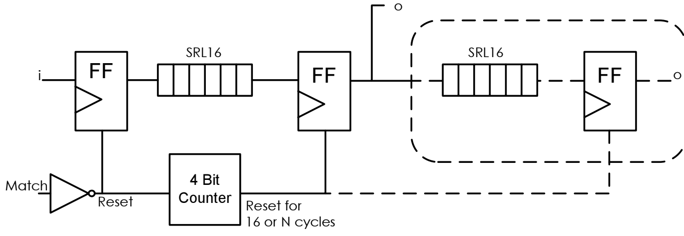

Fig. 2. The Exactly block: $a\{N\}$.

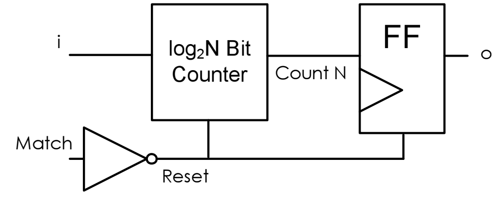

Fig. 3. The AtLeast block: $a\{N,\}$.

Kleene-star (*), Union (|) and Concatenation [3]. Based on those blocks, we have implemented blocks such as Caret, Dollar, Dot, Question-mark, Plus, etc. Furthermore, we introduce three new blocks to implement constrained repetitions (Exactly, AtLeast, and Between). Table II depicts the list with a brief description of our basic building blocks.

Concerning the constrained repetition blocks, our implementation minimizes the number of resources used, especially when comparing to previous DFA and NFA approaches [4]–[6], [13], [14], [16]. In the previous approaches, the constrained repetition blocks have to be unrolled, and thus require substantially larger amount of hardware resources.

Finally, we should note that our designs detect *all overlapping matches*, which is not the case for previous DFA approaches [5], [13], [14]. Consider the following example to illustrate overlapping matches. In case of the regular expression: "$((ad?|b)+bcd)|d(bb)?$" and the input stream: "$adbbcb$", the following overlapping matches are detected by our engines: "d", "dbb" and "$adbbcb$".

Exactly block: Figure 2 illustrates the Exactly block ($a\{N\}$). When a token (it represents the logic value '1', that is passed between the blocks, indicating they are allowed to try a match) is received, the exactly block forwards it after N matches. The circuit is implemented as follows: when a token occurs, it enters the shift register if there is a match of the a character (otherwise the register is reset). The shift register (successive FFs and SRL16 resources) is N bits long and one bit wide. The token is shifted for N cycles if there is no mismatch. In case of a mismatch, the shift register must be reset. Each SRL16 (16 bits long) is implemented in a single LUT and does not have a reset pin. Therefore, a mechanism is required to reset the contents of the shift register. To do so, we inserted flip-flops (FFs) between the SRL16s. The first FF is reset whenever a mismatch occurs. The rest of the FFs are reset for 16 cycles in order to erase the contents of the SRL16s. When the shift register is shorter than 17 bits ($N < 17$) then the reset of the second FF lasts $N-1$ cycles. We use a 4-bit counter in order to reset the FFs for 16 cycles. The use of SRL16 minimizes the cost of the block, since an SRL16 and a FF can be mapped on a single logic cell. As an example, the expression $a\{1000\}$ requires only 65 logic cells. Finally, notice that a new token can be immediately processed in the cycle after a reset, since the first FF and SRL16 continue to shift their contents. This is the reason why this block and all our basic building blocks detect overlapped matches.

AtLeast block: Figure 3 depicts the AtLeast block ($a\{N,\}$). When a token occurs, the block outputs a token after N matches. The output should remain active until the first mismatch. This is the actual reason why there is no need to record states for multiple intermediate tokens. Even if new tokens arrive after another token has already started matching, the output will not be affected. Thus, the AtLeast block can be implemented using only a counter that keeps track of the number of matches (up to N). About 72% of the constrained repetitions in Snort ruleset are of this kind. Therefore, the above implementation reduces substantially the area requirements of our hardware engines.

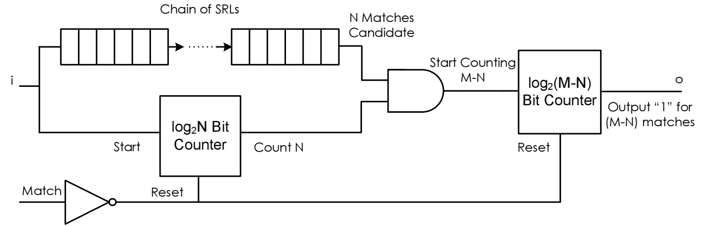

Fig. 4. The Between block: $a\{N,M\}$.

Between block: For the Between block ($a\{N,M\}$), the interval between N and M 'a' matches have to be detected. As depicted in Figure 4, the incoming token enters the shift register (length N) and starts/enables the first counter (counts up to N as long as there is no mismatch of a). After N simultaneous matches, if the output of the SRL16 is '1' and the counter has not been reset in the meantime, the second counter is enabled. The second counter (counts $M-N$) outputs '1' for $M-N$ simultaneous matches. Furthermore, it is reset and starts counting from '0' whenever it is enabled by the first counter, even if it has already started counting for a previous token. In case of an intermediate mismatch, the counters are reset. It could be assumed that the AtLeast block and the second counter would be sufficient to implement this block without the use of SRL16s. However, this is not possible since the intermediate tokens (i.e., tokens that arrive after a token starts matching until a mismatch or a match) would be lost and therefore other (overlapped) matches might be missed.

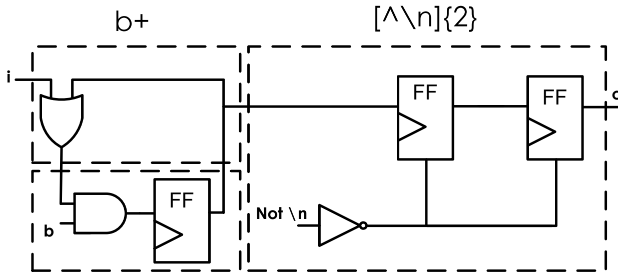

Fig. 5. An implementation for the regular expression $b^{+}[\hat{}\backslash n]\{2\}$.

We should note that the above constrained repetition blocks support repetitions of only a single character or a character class. They do not support repetitions of expressions that require more than one cycle to match (e.g., $(ab)\{10\}$), especially when the length of the expression between the parenthesis is unknown or not constant (e.g., $((ca)*|b)\{10\}$, $((ab|b)\{10\})$). In these cases, the expressions are unrolled. Fortunately, more than 95% of the constrained repetitions included in Snort regular expressions are of single character or character class.

We describe next an implementation example of the regular expression $b^{+}[\hat{}\backslash n]\{2\}$ illustrated in Figure 5. The above regular expression detects one or more "b" characters followed by two characters that are not "$new\ lines$". The module consists of a Plus block (upper-left), a character block (down-left), and an exactly$\{2\}$ block (on the right). Consider an input string "$bba \setminus n$". In the first cycle the input "i" will be high, and the first "b" will be accepted. Hence, the first FF will be activated. At the second cycle the second "b" will keep the first flip-flop high, and activate the second flip-flop. At the third cycle, an "a" arrives, the first flip-flop goes low, while the other two FFs are high and the module outputs a match for the input string "bba". Subsequently, an "$\backslash n$" character arrives, which resets the exactly block, and therefore, we do not have a second match for the input string '$ba \setminus n$".

B. Reducing Area

We apply several techniques to reduce the area cost of our designs. Apart from the centralized ASCII decoder, first introduced by Clark and Schimmel [6], we perform the following optimizations. As mentioned in the previous subsection, we employ the SRL16 modules to implement single bit shift registers requiring a single logic cell per shift register 17-bits long (one LUT and one flip-flop). Additionally, we share all the common prefixes of the ruleset so that regular expressions with a common prefix share the output of the same prefix sub-module. Static patterns and character classes are also implemented separately in order to share their results among the RegExp modules. The above optimizations, excluding the use of SRL16, save more than 30% of the total area resources. Below we discuss each optimization in more detail.

Xilinx SRL16: Usually, the states of the NFA are stored in flip-flops, each flip-flop representing a single state. An efficient solution to store certain states, which saves a lot of hardware resources, is to take advantage of the capability of Xilinx FPGAs to configure LUT resources as shift registers (SRL16s). Many basic blocks, such as constrained repetitions, need to store a large number of states, which can also be implemented by shift registers. Those shift registers are true FIFOs, and consequently, can be implemented with SRL16s which require a single logic cell (a single LUT plus a flip-flop) to store 17 states. This extensive use of SRL16s, to efficiently represent a great number of states, is one of the main optimizations to reduce the area of our designs.

Prefix Sharing: In Snort ruleset a large number of regular expressions have the same prefixes. Consequently, these prefixes can be shared as depicted in Figure 1. Without any additional hardware the common prefixes are implemented separately, as complete regular expressions, and their outputs provide an input to the suffixes of the corresponding regular expressions.

Character Classes sharing: Character Classes are widely used in Snort ruleset. Each character class is a Union of several characters. We implement these blocks separately and share their outputs in order to reduce the area cost. There are more than 3,500 character class cases in the Snort regular expressions, which are reduced to about 50 unique cases.

Static Patterns sharing: Similar to the character classes, we implement a static pattern matching module to match static strings included in the regular expression set. We utilize our previously proposed technique DCAM [7] and share the outputs of the module. The Snort v2.4 Regular Expressions set includes more than 1,500 static patterns (600 unique patterns of 7,600 characters in total), and therefore, a large amount of resources is saved.

C. Increase Performance

Two techniques have been employed to improve the performance of our designs. The first one keeps the fan-out of certain modules low, while the second one pipelines (when possible) combinational logic. More precisely, similarly to our previous work [18], we implemented fan-out trees to transfer the outputs of the decoder, the static pattern matching (DCAM) and the character class blocks to the regular expression modules. By doing so, we reduce the delays of the above connections at the cost of a few registers. Second, we pipeline modules such as the decoder, the DCAM and the character class modules. Pipelining the above modules is based on the observation that the minimum amount of logic in each pipeline stage can fit in a 4-input LUT and its corresponding register. This decision was made based on the structure of Xilinx logic cells. The area overhead cost of this pipeline is zero since each logic cell used for combinational logic also includes a flip-flop. Finally, the output of the pipelined modules is correctly aligned with the rest of the design.

D. Methodology - Hierarchical Implementation

We describe next the methodology followed to generate RegExp Engines (see Figure 6).

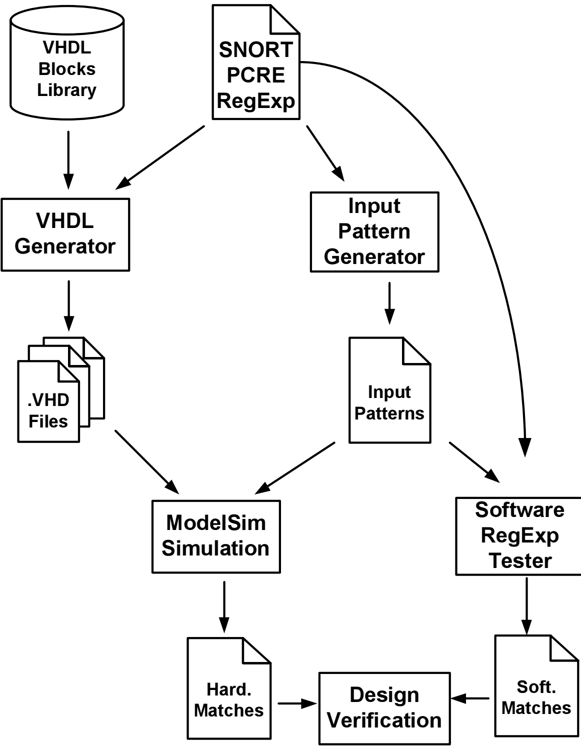

Fig. 6. Proposed methodology for generating regular expressions pattern matching designs.

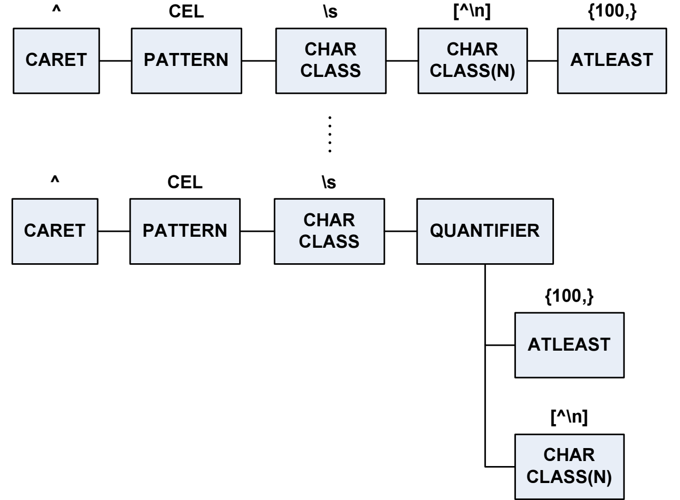

Fig. 7. Example of an hierarchical decomposition.

First, the regular expressions are extracted from the SNORT ruleset. Then, an automatic pre-processing step is responsible to rewrite regular expressions in order to discard any software related features (conditionals-lookahead) and to change other features (back references) to suit hardware implementation. A conditional-lookahead statement chooses, between multiple RegExp suffixes, a single one that should be followed, based on the condition. In hardware, we implement all the multiple RegExp suffixes and discard the conditional statement. A back-reference stores the string that matched a sub-RegExp and uses it in another part of the RegExp. We replace the back-references with the sub-RegExp they refer to. Therefore, our designs will not miss any matches compared to the PCRE-software implementation and might output some extra matches. Finally, the $flags$ included in regular expressions are considered, in order to change (if necessary) the functionality of some blocks (flags such as case (in)sensitive, multi-line, DOT includes $\backslash n$, etc.).

After rewriting, each regular expression is transformed into a list of tokens (in this case with the same meaning used by lexical analysis), and the sequences of tokens are assigned to "basic building blocks" which can be automatically mapped to hardwired modules. Performing multiple passes, our tool creates an hierarchical structure of each regular expression in order to generate the VHDL descriptions for the hardware engines. Figure 7 illustrates an example of an hierarchical decomposition of the regular expression "$\hat{}CEL\backslash s[\hat{}\backslash n]\{100,\}$". First, the tool parses the regular expression and creates the parenthesis hierarchy (upper part of Figure 7). Then, the parser detects and links the quantifiers (in this case the AtLeast block) and the Union operators as depicted in the bottom part of Figure 7.

Subsequently, the generation of the VHDL representation is straightforward. A bottom-up scheme is used to construct each regular expression pattern matching module based on the hierarchy extracted by the tool.

After the VHDL generation, the functionality of the design is automatically verified. Based on the regular expression set, our tool generates input traces to be used by the hardware implementations and by a software regular expression engine. As shown in Figure 6, the hardware implementations are verified by comparing their outputs with the results of the software RegExp engine. Finally, we should note that the compilation of the Snort RegExp into VHDL hardware description requires a few tens of seconds, while the place & route process of the design takes about a couple of hours. Consequently, the design can be updated/regenerated in acceptable time limits.

V. EVALUATION & COMPARISON

In this section, we evaluate the efficiency of our regular expression pattern matching engines and compare them against previous work. Our design has been implemented using Xilinx Virtex2 and Virtex4 devices (Xilinx ISE 8.1 software has been used). We measure performance in terms of processing throughput (post place & route results), and area cost in terms of required FPGA logic cells per non-Meta character[2]. We count non-meta characters to measure the density of the matching regular expressions. This might not be the most indicative metric to measure the size of a regular expression, however, it enables us to compare against related regular expression and static pattern matching approaches. Furthermore, in order to

[2]Meta characters are those characters which have a special meaning/function in the regular expression. The rest are the non-Meta characters. A character class $[A-Z]$ or a constrained repetition $a\{100\}$ counts as one non-Meta character.

TABLE III

CONSTRAINT REPETITIONS IN SNORT V2.4 PCRE REGULAR EXPRESSIONS.

	# of Constraint Repetitions in Snort v2.4	Average # of Iterations in Snort v2.4	Area (# of logic cells) # of Iterations 10 (-20)	100 (-200)	1,000 (-2,000)
Exactly - A{x}	209	185	8	12	65
AtLeast - A{x,}	470	956	7	15	25
Between - A{x,y}	2	124-238	16	37	106

compare in terms of area with designs that utilize memory, we measure the memory area cost based on the fact that 12 bytes of memory occupy area similar to a logic cell [19]. Finally, we evaluate our schemes and compare them with the related research, using a Performance Efficiency Metric (PEM), which takes into account both performance and area cost, described in the following equation:

$$PEM = \frac{Performance}{Area\ Cost} = \frac{Throughput}{\frac{Logic\ Cells\ +\ \frac{MEMbytes}{12}}{Non-Meta\ Characters}} \quad (1)$$

We implemented a Regular Expression Engine using the rules of the Snort v2.4 open-source intrusion detection system [1]. Snort v2.4 has 509 unique regular expressions of 19,580 non-Meta characters in total. Table III illustrates some statistics about the constrained repetitions of Snort regular expressions. Almost 700 of constrained repetitions were found, which on average define more than 700 iterations. Additionally, Table III shows the area cost of the constrained repetition blocks for various number of iterations. The exactly block of 1,000 iterations requires only 65 logic cells. The Atleast block of 1,000 iterations is more efficient since does not require SRL16s and occupies 25 logic cells. Finally, a Between block of 1,000-2,000 uses 106 logic cells.

TABLE IV

A COMPETE PAYLOAD SCANNER: MATCHING REGULAR EXPRESSIONS AND STATIC PATTERNS.

Description	Input bits /cycle	Device	Throughput (Gbps)	LUTs /FFs	Logic Cells	LC/char	MEM Kbits	# RegExp *or* Patterns
RegExp Engine Regular Expressions	8	Virtex2 -4000	2.000	16,720 / 19,656	25,074	1.39	0	509
`PHmem` Matching				4,165 / 8,252	9,466	0.28	630	2,188
Total	8	Virtex2 -4000	2.000		34,540		630	509+2,188

Our design processes one incoming byte per cycle and supports 2 Gbps throughput in a Virtex2-3000-6 and 2.9 Gbps in a Virtex4-40-12 device. The Xilinx post place & route report of the design indicates maximum clock frequencies of 250 MHz for Virtex2 and 362 MHz for Virtex4. The design requires about 25,000 logic cells to implement 509 regular expressions, conducing in this case to 1.28 logic cells per non-Meta character. Table IV depicts the detailed results of our RegExp Engine and also the implementation of a static pattern matching design using our previously proposed static pattern

TABLE V

COMPARISON BETWEEN OUR REGEXP ENGINE AND OTHER REGULAR EXPRESSION APPROACHES.

Description	RegExp/ Static Patterns[7]	Input bits /cycle	Device	Throughput (Gbps)	Logic Cells[6]	Logic Cells /char	MEM Kbits	#chars	PEM
Our Proposed Scheme	RegExp	8	Virtex2	2.0	25,074	1.28	0	19,580	1.56
			Virtex4	2.9					2.27
Lin *et al.* [13] NFA sharing sub-RegExp	RegExp	8	VirtexE -2000	N/A[4]	13,734	0.66	0	20,914	N/A[4]
Brodie *et al.* [14] DFAs	RegExp	64	Virtex2	4.0	860	N/A	8	per engine[5]	N/A[5]
			ASIC	16.0	∼247K[5]	22.2	2,296	11,126	0.66[5]
Baker *et al.* [15] DFA μ-controllers	RegExp	8	Virtex4 -100	1.4	N/A	2.56	6,000	16,715	0.22
Sidhu *et al.* [3] NFAs	RegExp	8	Virtex -100	0.46	1,920	66	0	29	0.01
Franklin *et al.* [4] NFAs	Static Patterns	8	VirtexE -2000	0.4	40,232	2.52	0	16,028	0.16
Clark *et al.* [6] Decoded NFAs	Static Patterns	8	Virtex2 -8000	2.0	29,281	1.70	0	17,537	1.19
		32		7.0	54,890	3.1	0		2.26
Moscola *et al.* [5] DFAs	Static Patterns	32	VirtexE -2000	1.18	8,134	19.4	0	420	0.06

matching technique Perfect-Hashing memory (`PHmem`) [8]. `PHmem` applies a perfect hashing technique to the incoming data in order to specify which static pattern (of the search pattern set) would possibly match and subsequently, read the pattern from a memory and compare it against incoming data. Hence, a payload scanner that matches *all* the regular expressions and the static patterns of the recent Snort ruleset needs about 34,500 logic cells and supports 2 Gbps, when processing one byte per cycle in a Virtex2.

In Table V, we attempt a fair comparison with previously reported research on *regular expression matching* designs. Apart from the performance and area results we calculate the PEM of the designs in order to measure and compare their efficiency. When compared against designs that process the same number of incoming bits per cycle, our designs achieve similar or better throughput. Furthermore, our RegExp engine requires the second lowest area cost. More precisely, compared to Lin *et al.* [13], our design requires about 2× more resources. However, Lin *et al.* do not report any performance results focusing only on minimizing the hardware resources of their NFA designs. Brodie *et al.* implemented DFAs using FSM-based engines aiming ASIC implementations [14]. Their design matches 315 Snort-PCRE regular expressions. However, due to their high area cost it cannot be implemented in current FPGA devices. An engine of Brodie *et al.* that matches approximately a single regular expression has been

[4] There are no performance results (frequency-throughput) for this design.

[5] The authors provide the logic and memory cost per Engine. They need 287 engines to match 315 PCRE-Snort regular expressions. Their complete ASIC design matching the 315 regular expressions (11,126 characters) would require about 247,000 logic cells and 27 Mbits of memory if it could be implemented in a Virtex2.

[6] Two *Logic Cells* form one *Slice*. We calculate the number of logic cells required for a design according to the next equation: $Logic\ Cells = 2 \times Slices$, where slices is the reported number of used slices of the Xilinx ISE tool.

[7] We denoted as “RegExp” the designs that match PCRE Snort regular expressions, and “Static patterns” the ones that match IDS (Snort) static patterns by converting them into regular expressions.

prototyped in a Virtex2 device. It achieves 4 Gbps (2× vs. our design), processing 8 bytes per cycle. A single engine requires 860 logic cells and 8 Kbits memory. The area cost of their complete ASIC design (if calculated in terms of logic cells) is 19× higher than our approach, while their performance is 5.5× higher. Their ASIC design, calculating the area cost in FPGA terms (logic cells), is 3.4× less efficient than our Virtex4 implementation. Baker *et al.* implemented multiple DFA microcontrollers, which are updated by changing the contents of their memories instead of reconfiguring the FPGA device [15]. Due to this decision, their design requires about 5× more resources than our design. Furthermore, their design achieves about half the throughput compared to our solution and its efficiency is 10× lower.

Clark *et al.* and Franklin *et al.* match only static patterns transformed into regular expressions [4], [6] and therefore their designs are simpler. Compared to Franklin *et al.* we achieve more than 2× their throughput (taking into account that VirtexE devices are about 30-40% slower than Virtex2) and occupy about half the area. Compared to Clark and Schimmel design that processes 8-bits per cycle, we achieve similar performance requiring 25% fewer resources. Our design is about 30% less efficient compared to Clark and Schimmel second design (processes 32 bits per cycle). In static pattern matching, it is relatively straightforward to exploit parallelism and to increase resource sharing. Notice, however, this shows that our designs, albeit dealing with dynamic pattern matching, are also comparable to static pattern matching solutions (unable to deal with most RegExp).

Finally, Sidhu *et al.* and Moscola *et al.* implemented only few regular expressions. Therefore, their results may not be compared to designs that match complete rulesets. Even though, our approach clearly outperforms their designs.

VI. Conclusions

This paper proposed a Regular Expression pattern matching Engine for Snort intrusion detection system. We presented a method to generate hardwired engines that match Perl-compatible regular expressions. Our method uses new basic building blocks to implement constrained repetitions and several techniques to minimize the area cost and to improve performance. Furthermore, we discussed our methodology and suggested techniques to rewrite PCRE regular expressions in order to suit hardware implementations. Concerning the entire Snort regular expression set, our approach permits to achieve a throughput of 2 and 2.9 Gbps using Virtex2 and Virtex4 devices, respectively. It requires 1.28 logic cells per non-Meta character. Based on the performance efficiency metric (PEM), our design is 10 and 3.4 × more efficient than the best related FPGA and ASIC approaches, respectively. Even compared to designs of the same datapath width (8-bits) that match static patterns using regular expressions, and therefore are simpler, our approach is 20% to 10× more efficient. Future work will focus on researching building blocks to support additional regular expressions constructs and on evaluating more advanced forms of resource sharing by identifying common subexpressions (note that currently our approach already shares common prefixes).

References

[1] SNORT official web site, "http://www.snort.org."

[2] M. Fisk and G. Varghese, "An Analysis of Fast String Matching Applied to Content-based Forwarding and Intrusion Detection," in *Techical Report CS2001-0670*, University of California - San Diego, 2002.

[3] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[4] R. Franklin, D. Carver, and B. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[5] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a Content-Scanning Module for an Internet Firewall," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.

[6] C. R. Clark and D. E. Schimmel, "Scalable Parallel Pattern-Matching on High-Speed Networks," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004.

[7] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004.

[8] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A Reconfigurable Perfect-Hashing Scheme for Packet Inspection," in *Proceedings of 15th Int. Conf. on Field Programmable Logic and Applications*, 2005.

[9] G. Papadopoulos and D. Pnevmatikatos, "Hashing + Memory = Low Cost, Exact Pattern Matching," in *15th International Conference on Field Programmable Logic and Applications*, 2005.

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

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

[12] R. McNaughton and H. Yamada, "Regular Expressions and State Graphs for Automata," *IEEE Transactions on Electronic Computers*, vol. 9, pp. 39–47, 1960.

[13] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of regular expression pattern matching circuits on FPGA," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, 2006, pp. 12–17.

[14] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 191–202.

[15] Z. K. Baker, H.-J. Jung, and V. K. Prasanna, "Regular Expression Software Deceleration For Intrusion Detection Systems," in *16th International Conference on Field Programmable Logic and Applications*, 2006.

[16] P. Sutton, "Partial Character Decoding for Improved Regular Expression Matching in FPGAs," in *In Proceedings of IEEE International Conference on Field-Programmable Technology (FPT)*, 2004, pp. 25–32.

[17] PCRE -Perl Compatible Regular Expressions, "http://www.pcre.org/."

[18] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System," in *13th International Conference on Field Programmable Logic and Applications*, 2003.

[19] T. Sproull, G. Brebner, and C. Neely, "Mutable Codesign For Embedded Protocol Processing," in *Proceedings of 15th International Conference on Field Programmable Logic and Applications*, 2005.