Regular Expression Matching in Reconfigurable Hardware

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

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

[♮] *INESC-ID Lisboa, Portugal*
joaobispo@gmail.com, jmpc@acm.org

[*] *Department of Informatics Engineering, IST/UTL, Lisboa, Portugal*

***Abstract*— In this paper we describe a regular expression pattern matching approach for reconfigurable hardware. Following a Non-deterministic Finite Automata (NFA) direction, we introduce three new basic building blocks to support constraint repetitions syntaxes more efficiently than previous works. In addition, a number of optimization techniques are employed to reduce the area cost of the designs and maximize performance. Our design methodology is supported by a tool that automatically generates the circuitry for the given regular expressions and outputs HDL (Hardware Description Language) representations ready for logic synthesis. The proposed approach is evaluated on network Intrusion Detection Systems (IDS). Recent IDS use regular expressions to represent hazardous packet payload contents. They require high-speed packet processing providing a challenging case study for pattern matching using regular expressions. We use a number of IDS rulesets to show that our approach scales well as the number of regular expressions increases, and present a step-by-step optimization to survey the benefits of our techniques. The synthesis tool described in this study is used to generate hardware engines to match 300 to 1,500 IDS regular expressions using only 10-45K logic cells and achieving throughput of 1.6-2.2 and 2.4-3.2 Gbps on Virtex2 and Virtex4 devices, respectively. Concerning the throughput per area required per matching non-Meta character, our hardware engines are 10-20× more efficient than previous FPGA approaches. Furthermore, the generated designs have comparable area requirements to current ASIC solutions.**

I. Introduction

Many applications in several fields, such as biomedical, data mining, and network processing, employ regular expressions to describe search patterns. Biomedical applications use regular expressions for biosequence search [1]–[3], i.e., in DNA matching, protein matching or genomes search. The exponential growth of their biosequence databases greedily imposes high-performance demands. Networking systems also need high-speed regular expression pattern matching for content-based packet processing [4], [5]. For example, regular expressions are used in network security (e.g., intrusion detection systems), to describe known attack patterns [17] or in traffic management and routing where packets are classified and processed upon their content. In many cases, such as the above, regular expression pattern matching needs to support high processing throughput at the lowest possible hardware cost.

When performance is critical, software platforms may not be able to provide efficient regular expression implementations. It is a fact that they can be more than an order of magnitude slower than hardware implementations, their performance does not scale well as the number of regular expressions increases and their memory requirements may be substantially large [4]–[7]. Reconfigurable systems (e.g., Field Programmable Gate Arrays) may provide an efficient solution for high speed regular expression pattern matching. Field Programmable Gate Arrays (FPGAs) can operate at hardware speed and exploit parallelism. Moreover, they provide the required flexibility to change the regular expression ruleset implementation on demand. As the size of the regular expressions set grows, conventional CPU performance may deteriorate appreciably compared to an FPGA-based approach. Consequently, FPGAs offer an excellent implementation platform for regular expression pattern matching. Architectures such as the Molen [8] or the ones described in [9] can be followed to best exploit the advantages of reconfigurable hardware.

Given an input string $T[1..n]$ which uses a finite set of symbols $\sum$ (alphabet) and a regular expression R of the same alphabet which describes a set of strings $S(R) \subseteq \sum*$, then matching the regular expression R is to determine whether $T \in S(R)$. For decades, significant effort has been put on implementing regular expressions in software. The Non-deterministic Finite Automata (NFA) approaches have limited performance in software due to their multiple active states. Consequently, Deterministic Finite Automata (DFA) are usually adopted. DFAs allow only one active state at a time, suit better the sequential nature of General Purpose Processors and achieve higher performance. However, DFAs suffer from state explosion [10], especially when regular expressions contain wildcards ('.', '?', '+', '*'), character classes or constraint repetitions. 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 alphabet, i.e., 2^8 symbols for the extended ASCII code), while an NFA representation would require only $O(n)$ states [11]. Several studies manage

to increase the performance of DFAs in software and reduce the required number of states [4]–[7]. However, this is not always possible and usually compromises the accuracy of the implementations (i.e., ignoring overlapping matches).

Alternatively, regular expressions can be implemented in hardware. A variety of solutions have been proposed and implemented in technologies that range from Programmable Logic Arrays [12], [13] to FPGAs [14]. In the past, some basic blocks have been introduced to implement Wildcards, Union and Concatenation regular expression operators in reconfigurable hardware [15]. However, more complicated regular expression syntaxes are not efficiently supported. For example, in order to implement constraint repetitions, the same circuit has to be repeated for a number of times equal to the number of repetitions. When a DFA approach is chosen, a substantially larger number of states is required compared to NFA solutions. As a consequence DFA hardware engines result in inefficient designs in terms of area (logic and/or memory). On the other hand, when implemented properly, NFAs can be more compact and area efficient; hardware is inherently concurrent, and therefore can be suitable for NFA implementations.

In this paper we present an NFA-based approach to match multiple regular expressions in reconfigurable hardware. We apply and evaluate our approach in IDS rulesets. The main contributions of this work are the following:

- We introduce three new basic building blocks for *constraint repetition* operators, which are able to detect all overlapping matches. These blocks handle regular expressions repetitions that require a single cycle to match. When combined with previous research in NFA-based hardware implementations, efficient designs can be achieved.
- Theoretical proofs are presented to show that two of the constraint repetition blocks can be simplified without affecting their functionality.
- To improve the efficiency of the designs, we insert a pre-processing optimization stage. The extracted regular expressions are modified to suit our hardware implementation. Syntax features that only facilitate software implementations are discarded while others are replaced by equivalent ones (i.e., conditional branches, lookahead statements).
- We employ several techniques to reduce the area requirements of our designs, such as regular expressions prefix sharing, pre-decoding, centralized static pattern matching and blocks of character classes, etc. Furthermore, we take advantage of the Xilinx SRL16 shift registers to store multiple states using fewer FPGA resources.
- A methodology is introduced to automatically generate the regular expression pattern matching engines from the IDS rulesets. We show how a hierarchical representation of the regular expressions is used to facilitate the automatic VHDL generation using basic building blocks. A tool that outputs the VHDL circuit description of the design has been developed.
- We are able to generate efficient regular expression engines, in terms of area and performance, outperforming previous FPGA-based approaches. Our designs match over 1,500 regular expressions and support 1.6-3.2 Gbps throughput requiring a few tens of thousand logic cells. Finally, the area requirements are comparable with DFA-based ASIC implementations which suffer however from state explosion.

The remainder of this paper is organized as follows. In Section II we briefly discuss IDS characteristics and their Perl-compatible regular expression syntax (PCRE) [16], while in Section III we survey previous work on hardware regular expression pattern matching. Section IV describes the top-level approach of our regular expression engines, the basic building blocks and the techniques employed to reduce area and increase performance. Section V presents the methodology followed to automatically generate VHDL code describing the regular expression hardware engine for a given set of regular expressions. In Sections VI and VII, we present the implementation results of our designs and compare them with related work. Finally, Section VIII draws some conclusions and suggests future work.

II. Intrusion Detection & Perl-compatible Regular Expressions

High speed and always-on network access is becoming commonplace around the world, creating a demand for increased network security. Network Intrusion Detection Systems (IDS) such as Snort [17] and Bleeding Edge [18] 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. In the past years, many researchers have worked on reconfigurable hardware solutions for IDS focusing mostly on the payload scan, which turns out to be the most computationally intensive task [19]. Numerous techniques for reconfigurable IDS static pattern matching have been proposed [14], [20]–[25]. Many of them employ regular expressions to represent the static search patterns, implementing either non-deterministic or deterministic finite automata (NFAs or DFAs) [20]–[22]. However, recent network intrusion detection systems use more extensively regular expressions instead of static patterns to represent more efficiently hazardous packet payload contents. These regular expressions attack descriptions need to be matched at high-speed against incoming traffic.

Regular expressions, and especially their complex features such as constraint repetitions, may create a significant bottleneck for IDS performance. Table I illustrates the recent increase of regular expressions in Snort [17], [26] and Bleeding Edge [18] IDS rulesets along with the static patterns included in these sets. Additionally, the exact number of constraint repetitions is reported for each ruleset. Constraint repetitions are operators which indicate a sub-expression to be matched repeatedly for a defined number of repetitions (Exactly, AtLeast, and Between quantifiers, e.g., $a\{10\}$, $a\{10,\}$, $a\{10,12\}$). IDS rulesets include a significant number of regular expressions and constraint repetitions which continuously increases. For example, in May 2003 only 65 regular expressions were used, in April 2006 increased to more than 500 and within the year tripled exceeding 1,500. It is expected that the number

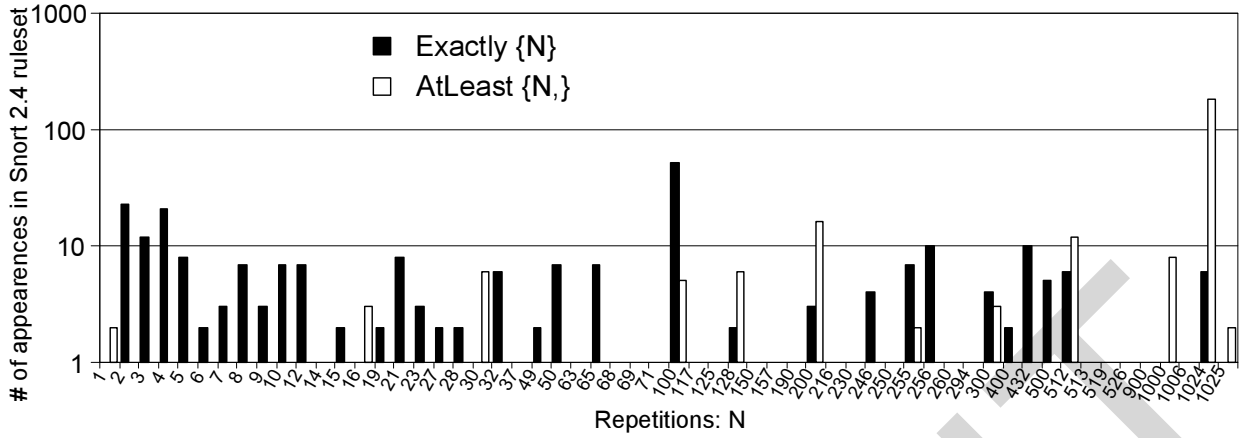


Fig. 1. Distribution of two of the most commonly used constraint repetitions in Snort IDS, type Exactly and AtLeast. Results are for the Snort v2.4 Oct. 2006 version.

TABLE I
REGULAR EXPRESSIONS AND STATIC PATTERNS USED IN SNORT AND BLEEDING EDGE RULESETS.

Rulesets	#Static patterns	Regular Expressions			
		Total	Constraint Repetitions		
			#Exactly	#AtLeast	#Between
Snort 2.4 (Jan. 2007)	3,432	1,615	274	495	11
Snort 2.4 (Dec. 2006)	3,377	1,589	273	495	10
Snort 2.4 (Nov. 2006)	3,391	1,616	271	495	10
Snort 2.4 (Oct. 2006)	3,248	1,504	265	478	11
Snort 2.4 (Apr. 2006)	1,537	509	209	470	2
Snort 2.3 (Mar. 2005)	2,188	301	124	464	1
Snort 2.2 (July 2004)	1,042	157	85	22	1
Snort 2.1 (Feb 2004)	942	104	52	19	0
Snort 1.9 (May 2003)	909	65	46	1	0
Bleeding (Dec. 2006)	968	318	47	7	17
Bleeding (Nov. 2006)	968	317	48	7	17
Bleeding (Oct. 2006)	934	310	43	7	17

of regular expressions in the IDS rulesets will continue to increase since new attack descriptions are constantly added to the rulesets. Based on the data present at the moment, the number of regular expressions seems to increase faster than the static patterns in Snort v2.4 (within 2006, static patterns increased 2.2× and regular expressions 3×). Figure 1 illustrates the number of repetitions and the number of appearances of the most common constraint repetitions (Exactly{N} and AtLeast{N,}) for the Snort v2.4 ruleset (Oct. 2006 version). Such operations appear tens or even hundreds of times having up to a thousand repetitions, which indicates current IDS regular expressions complexity. On average, one constraint repetition per two regular expressions exists in Snort 2.4. Converting them to DFAs would result in thousands of states, which would require a significant number of hardware resources for encoding. Consequently, dedicated blocks for these operations would substantially reduce the cost of the IDS regular expression implementations.

Snort and Bleeding Edge IDS adopted the Perl-compatible regular expression syntax (PCRE) [16]. For example, `alert tcp any -> (pcre:"/^PASS\s*\n/smi";)` is a Snort rule, it detects any packet containing a payload string which matches the "`/^PASS\s*\n/smi`" regular expression. Apart from the well known features of the strict definition of regular expressions, PCRE is extended with new operations such as flags and constraint repetitions. Table II describes the PCRE basic syntax supported by our regular expression pattern matching engines. There are two types of features that are supported. The first ones are directly mapped to hardware building blocks (wildcards, union, concatenation, constraint repetitions, and character classes) and are explained in more detail in section IV. The second type is supported by replacing them during a pre-processing stage with equivalent expressions that suit our hardware implementations (backslash to escape meta-characters, dollar, flags, etc.). 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), and a "continue from the previous match" command (\G). Since current Snort and Bleeding Edge rulesets do not use these features, our synthesis tool is able to generate designs matching all the regular expressions of the IDS rulesets.

III. RELATED WORK

In 1959, Rabin and Scott introduced the non-deterministic finite automata (NFAs) and the concept of non-determinism [27], showing that NFAs can be simulated by (potentially much larger) DFAs in which each DFA state corresponds to a set of NFA states. McNaughton and Yamada [28] and Thompson [29] described two of the first methods to convert regular expressions into NFAs. Thompson encodes the selection of state transitions with explicit choice nodes and unlabeled arrows (ϵ-transitions). On the other hand, McNaughton and Yamada, avoided unlabeled arrows and allowed instead NFA states to have multiple outgoing arrows with the same label. Their method can be easier directly mapped in hardware, since each transition "consumes" an incoming character and the number of states is reduced.

Matching Regular Expressions in hardware has been widely studied in the past. In 1979, Mukhopadhyay proposed the basic blocks for Concatenation, Kleene-star and Union operators [15]. In 1982, Floyd and Ullman discussed the implementation

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

Feature	Description
a	All ASCII characters, excluding meta-characters, match a single instance of themselves
[\^$.—?*+()	Meta-characters. Each one has a special meaning
.	Matches any character except new line
\?	Backslash escapes meta-characters, returning them to their literal meaning
[abc]	Character class. Matches one character inside the brackets. In this case, equivalent to (a|b|c)
[a-fA-F0-9]	Character class with range.
[^abc]	Negated character class. Matches every character except each non-Meta character inside brackets.
RegExp*	Kleene Star. Matches zero or more times the regular expression.
RegExp+	Plus. Matches one or more times the regular expression.
RegExp?	Question. Matches zero or one times the regular expression.
RegExp{N}	Exactly. Matches N times the regular expression.
RegExp{N, }	AtLeast. Matches N times or more the regular expression.
RegExp{N,M}	Between. Matches between N and M times the regular expression.
\xFF	Matches the ASCII character with the numerical value indicated by the hexadecimal number FF.
\000	Matches the ASCII character with the numerical value indicated by the octal number 000.
\d, \w and \s	PCRE Shorthand character classes matching digits 0-9, word characters (letters and digits) and whitespace, respectively.
\n, \r and \t	Match an LF character, CR character and a tab character, respectively.
(RegExp)	Groups regular expressions, so operators can be applied.
RegExp1RegExp2	Concatenation. Regular Expression 1, followed by Regular Expression 2
RegExp1 | RegExp2	Union. Regular Expression 1 or Regular Expression 2.
^RegExp	Matches Regular Expression 1 only if at the beginning of the string.
RegExp$	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.

of NFAs in Programmable Logic Arrays [12], proposing among other aspects a hierarchical implementation described by the McNaughton-Yamada algorithm [28]. Foster, described some regular expressions modifications to avoid latch formation in regular expressions implementation [30]; for example, two kleene-stars when put in sequence can form a latch.

More recently, reconfigurable hardware proved to be beneficial for regular expression matching. FPGAs can provide hardware speed, high degree of parallelism and the flexibility to modify the functionality of a design on demand. Consequently, FPGA devices may offer a high-speed regular expressions pattern matching of large sets and permit to modify and update the hardware engines according to the IDS ruleset.

Several NFA implementations have been proposed for reconfigurable hardware. In 1999, Sidhu and Prasanna presented NFA-based implementations of regular expressions in FPGAs [14] and used the basic blocks of [15] for Concatenation, Kleene-star and Union operators. Hutchings $et\ al.$ used NFAs to represent all the Snort static patterns into a single regular expression, requiring substantially lower area [20]. Clark and Schimmel used pre-decoding to share the character comparators of their NFA implementations and thus reducing even more hardware resources [22], [31]. Lin $et\ al.$ saved area resources of their NFA designs by sharing parts of the regular expressions [32]. Finally, Moscola $et\ al.$ in [33] attempted to combine previous NFA approaches [14], [22] with a "pre-decoding" static pattern matching technique [23], [34].

Despite the fact that FPGAs are suitable for NFAs, several researchers followed a DFA direction. 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 [21]. More recently, Baker $et\ al.$ described a microcontroller DFA implementation in FPGA for matching IDS regular expressions [35]. Their design updates its ruleset by only changing the memory contents. IDS regular expressions are converted to DFAs in order to be ported into the proposed microcontroller.

Brodie $et\ al.$ proposed an ASIC implementation of regular expressions in [36]. They converted the IDS patterns and regular expressions into DFAs and implemented them in high-speed FSM structures specially designed for regular expression matching. Their architecture uses memories to store transition and indirection tables and therefore the regular expressions can be modified by changing the contents of the memory blocks.

In summary, some researchers use DFAs to evaluate regular expressions resulting in designs with significant area/memory requirements [21], [35], [36]. The rest employ NFAs, however, they do not solve the problem of constraint repetitions and consequently, as Sutton notes in [37], need to repeat the same circuit in order to support them (i.e., fully unrolling the constraint repetitions). This work attempts to circumvent disadvantages and bottlenecks of previous approaches and also shows a methodology to automatically generate regular expression hardware engines. Such methodology has been implemented in a synthesis tool and can be applied to large sets of regular expressions.

IV. REGULAR EXPRESSIONS ENGINE

In this section, our regular expression engine is described.We exploit reconfigurable hardware and generate specialized circuitry for any given set of regular expressions. Figure 2 depicts the top-level diagram of the proposed regular expressions pattern matching engine. The incoming data (one byte per cycle) 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, each character is matched only once and all the regular expression modules receive the output lines from the decoder. For each regular expression there is a separate module. Regular expressions with common prefixes share the same prefix sub-module. The static sub-patterns (more than one character long) included in each regular expression are matched separately in a DCAM (Decoded CAM) static pattern matching module described

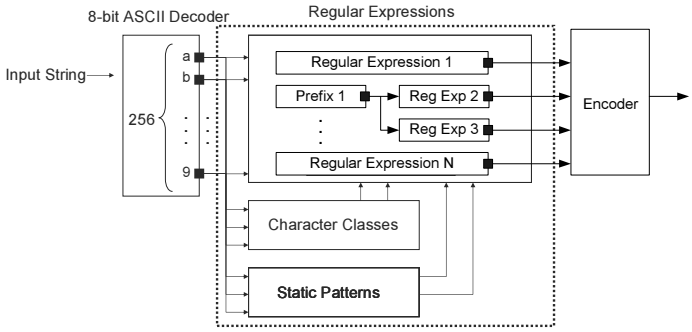


Fig. 2. Block diagram of our Regular Expression Engines.

in our previous work [23]. Similarly, the character classes (union of several characters e.g., $(a|b)$) are also implemented separately and share their results among the regular expression modules. Both static pattern matching and character class modules are fed from the ASCII decoder. Each regular expression module outputs a match for the corresponding regular expression and subsequently, all the matches are encoded on a priority encoder described in [38].

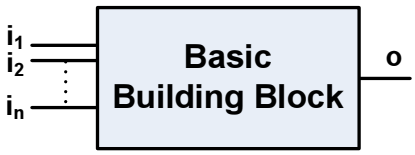


Fig. 3. Generic description of a basic building block.

A. Basic NFA blocks

The proposed design is based on building blocks that implement basic regular expression syntax features. Figure 3 illustrates a generic view of a basic building block. It consists of an output o and one or many (e.g., in the Union block) inputs i (input tokens). The decoded characters, pattern matching and character classes signals can be considered as input tokens. Table III depicts the list of all the supported blocks along with a brief description. For Kleene-star (*), Union (|) and Concatenation we use the blocks described by Mukhopadhyay [15]. Extending upon them we implement blocks for Caret, Dollar, Dot, Question-mark, Plus, etc. Three new blocks are introduced and described below to implement constraint repetitions (Exactly, AtLeast, and Between).

Concerning the constraint repetition blocks, our implementation minimizes the number of required resources, when compared to previous DFA and NFA approaches [20]–[22], [32], [36], [37]. In the previous approaches, the constraint repetition blocks have to be fully unrolled, and thus require significant amount of hardware resources.

We should further note that our designs detect *all* overlapping matches, which is not the case for previous DFA approaches [21], [32], [36]. To exemplify overlapping matches consider the following: given the regular expression "$((ad?|b) + bcd)|d(bb)?$" and the input stream "*adbbcb*", the following overlapping matches should be detected "*d*", "*dbb*" and "*adbbcb*".

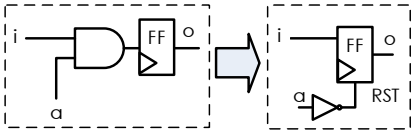
TABLE III
THE BASIC BUILDING BLOCKS OF OUR REGULAR EXPRESSION ENGINE.

Block	Description	Non Meta character count
Character	Matches a single character, based on the design of single character described in [15].	1
Union	Union operator of the regular expressions r_i, as described in [15].	The non meta chars of the Regular Expressions r_i
Concatenation	Concatenation operator of the regular expressions r_i, as described in [15].	The non meta chars of the Regular Expressions r_i
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.	pattern length
Dollar ($)	Validates the match if in the end of the packet/string. Based on the Character Block [15].	0
Dot	Matches any character except the new line. Based on the Character Block [15] the input character is the "$newline$" ($\backslash n$) character inverted.	1
Caret (^)	Starts a match every time a packet/string arrives. Based on the Character Block [15], the input character is the "$beginning\ of\ packet$" character.	0
Character Class	Matches a set of characters. Based on the Character Block [15], the input character is one of the outputs of $character\ class$ module. The $character\ class$ module ORs the characters included in a character class.	1
RegexBlock	Encapsulates hardware blocks that implement regular expressions or sub-blocks of regular expressions.	# of non MetaChars of the RegExpr
Question (?)	$r?$, One or zero times the regular expression r, based on the design of Kleene-star ($r*$) described in [15]. The incoming OR gate (to the flip-flop) has to be removed, consequently, the input token (i) goes directly to the flip-flop.	# of non MetaChars of the RegExpr r
Plus (+)	$r+$, One or more times the regular expression r, based on the design of Kleene-star ($r*$) described in [15]. 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.	# of non MetaChars of the RegExpr r
Kleene (*)	$r*$, Zero or more times the regular expression r, as described in [15].	# of non MetaChars of the RegExpr r
Exactly	$r\{N\}$, Matches r exactly N times. Constraint Repetition for single characters and sets of characters. Described in Section IV-A.	# of non MetaChars of the repeated RegExpr r
AtLeast	$r\{N,\}$, Matches r at least N times. Constraint Repetition for single characters and sets of characters. Described in Section IV-A.	# of non MetaChars of the repeated RegExpr r
Between	$r\{N,M\}$, Matches r between N and M times. Constraint Repetition for single characters and sets of characters. Described in Section IV-A.	# of non MetaChars of the repeated RegExpr r

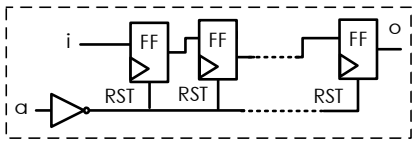
Exactly block: This block (e.g., $a\{N\}$) will report a match for each N successive 'a' symbols. The Exactly block $a\{N\}$ is actually the concatenation of N characters 'a' and can be defined as follows:

$$a\{N\} = \begin{cases} \epsilon & \textbf{for } N = 0 \\ a & \textbf{for } N = 1 \\ aa..a, \mathbf{n\ times} & \textbf{for } N > 1 \end{cases} \quad (1)$$

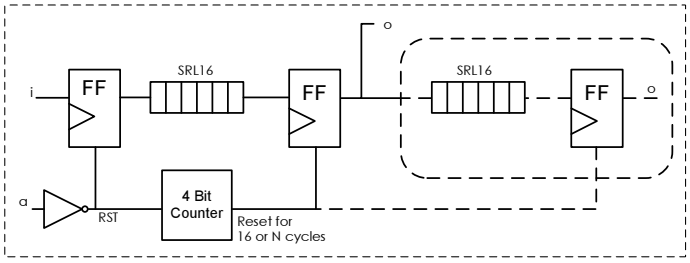
Figure 4(a) depicts the circuit that matches a single character a; it is a logical AND between the input i and the match of character a feeding a flip-flop. This circuit can be reduced to a single flip-flop having i as an input and the $\bar{a}$ as a reset.



(a) a{1} = a



(b) a{N} = aa...a, n times



(c) The proposed Exactly block: $a\{N\}$. Successive flip-flops and SRL16s with a reset mechanism.

Fig. 4. The Exactly block: $a\{N\}$.

Applying the concatenation for N a's results in a sequence of flip-flops as depicted in Figure 4(b). The correctness of this circuit can be proven by induction, however, is also given by the definition of the concatenation function and therefore omitted from this paper. The sequence of flip-flops to implement $a\{N\}$ is actually a true FIFO with a reset (flush) pin, and can be designed for FPGA-based platforms as depicted in Figure 4(c).

The proposed Exactly block (Figure 4(c)) has the following functionality. When a token i is received in the input, the exactly block forwards it after N matches. The input token enters the shift register if there is a match of the 'a' character (otherwise the register is reset). The shift register (successive flip-flops 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, flip-flops are inserted between the SRL16s. The first flip-flop is reset whenever a mismatch occurs. The rest of the flip-flops are reset for 16 cycles in order to erase the contents of their previous SRL16. When the shift register is shorter than 17 bits ($N < 17$) then the reset of the second flip-flop lasts $N - 1$ cycles. We use a 4-bit counter in order to reset the flip-flops for 16 cycles. It is noteworthy that a new token can be immediately processed in the cycle after a reset, since the first flip-flop and SRL16 continue to shift their contents. The block can keep track of all incoming tokens and therefore supports overlapping matches. The exactly block has an area cost $O(N)$. However, the use of SRL16 minimizes the actual resources, since an SRL16 and a flip-flop can be mapped on a single logic cell. The implementation cost in terms of logic cells is relatively low, for example, the regular expression $a\{1000\}$ requires only 63 logic cells.

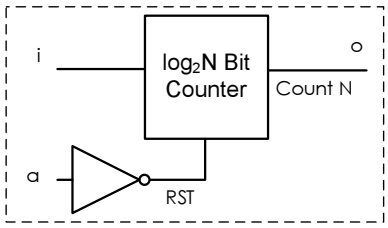


Fig. 5. The AtLeast block: $a\{N,\}$.

AtLeast block: In this block (e.g., $a\{N,\}$) continuous matches will be reported for each N or more successive 'a' symbols. When a token is received, the block should output a token after N matches and the output should remain active until the first mismatch. The AtLeast block can be defined as:

$$a\{N,\} = \bigcup_{k=N}^{\infty} a\{k\} \tag{2}$$

We prove next that the output of the AtLeast block is affected only by the first input token after the last reset, while subsequent tokens can be ignored. Consequently, we can implement this block with a single counter controlled by the first token received after a reset (Figure 5). The counter counts up to N and remains at value N activating the output until a mismatch.

Theorem 1: The output of the AtLeast block $a\{N,\} = \bigcup_{k=N}^{\infty} a\{k\}$ depends on only the first still active input token (received after the last mismatch). Any subsequent input token does not affect the output of the block.

Proof: Let i_{last} be the last token received at time $t = 0$, then the output of the AtLeast block for this token is:

$$AtLeast(i_{last}) = \bigcup_{k=N}^{\infty} a\{k\} \tag{3}$$

Let also i_{first} be the first token (still processed, not reset) received at time $-t < 0$. Then the remaining AtLeast output for i_{first} is:

$$AtLeast(i_{first}) = \begin{cases} \bigcup_{k=N-t}^{\infty} a\{k\} & \textbf{for } N > t \\ \bigcup_{k=0}^{\infty} a\{k\} & \textbf{for } N \leq t \end{cases} \tag{4}$$

However, $AtLeast(i_{last}) \subset AtLeast(i_{first})$ and therefore i_{last} can be ignored. ■

Hence, the AtLeast block can be implemented using a single counter controlled by the first input token after a reset. The counter keeps track of the number of matches (up to *N*) and its implementation cost is $O(log_2 N)$. About 70% of the constraint repetitions in Snort v2.4 are of this kind. Therefore, the above implementation substantially reduces the area requirements of the hardware engines.

Between block: The Between block (e.g., $a\{N, M\}$), matches N to M successive matches of 'a', its formal definition is the following:

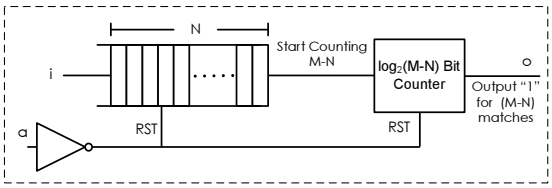


Fig. 6. The Between block: $a\{N, M\}$ = $a\{N\}a\{0, M-N\}$.

$$a\{N, M\} = \bigcup_{k=N}^{M} a\{k\} \tag{5}$$

Let us first define a block $a\{0, N\} = \bigcup_{k=0}^{N} a\{k\}$ which has an active output from the time an input token is received up to N matches. We prove next that the output of the $a\{0, N\}$ block is affected by only the last input token, while previous tokens can be ignored. Consequently, this block can be implemented by a single counter which resets at every mismatch, starts counting from '0' every time a new input token i arrives, counts up to N and then resets.

Theorem 2: The output of the block $a\{0, N\} = \bigcup_{k=0}^{N} a\{k\}$ depends on only the last still active input token (received after the last mismatch). Any previous input token does not affect the output of the block.

Proof: Let i_{last} be the last token received at time $t = 0$, then the output of the $a\{0, N\}$ block for this token is:

$$a\{0, N\}(i_{last}) = \bigcup_{k=0}^{N} a\{k\} \tag{6}$$

Let also i_{prev} be any previous token still active received at time $-t < 0$, then the remaining output tokens of the $a\{0, N\}$ block for i_{prev} is:

$$a\{0, N\}(i_{prev}) = \begin{cases} \bigcup_{k=0}^{N-t} a\{k\} & \textbf{for } N > t \\ \emptyset & \textbf{for } N \leq t \end{cases} \tag{7}$$

However, $a\{0, N\}(i_{prev}) \subset a\{0, N\}(i_{last})$ and therefore i_{prev} can be ignored. ■

The Between block $a\{N, M\}$ can be considered as the concatenation of an exactly block $a\{N\}$ and a block such the one described above $a\{0, M-N\}$. As depicted in Figure 6, the proposed design for the Between block is actually $a\{N\}a\{0, M-N\}$. The functionality of the Between block is the following. The incoming token enters the shift register (length N) which can be reset (flushed) by a mismatch. After N simultaneous matches, the shift register outputs '1' and the counter is enabled. The counter (counts up to $M-N$) outputs '1' for $M-N$ simultaneous matches. Furthermore, it is reset and starts counting from '0' whenever it is enabled by the shift register, even if it has already started counting for a previous token. In case of an intermediate mismatch, the counter is reset. It could be assumed that the $a\{0, M-N\}$ block and a second counter (replacing the $a\{N\}$) would be sufficient to implement this block without the use of the shift register. However, this is not possible since the intermediate tokens would be lost and therefore other (overlapping) matches would be missed. Consequently, the implementation cost of the between block is $O(N + log_2(M-N))$, and like the exactly block due to the use of SRL16s the area requirements are not high in terms of logic cells.

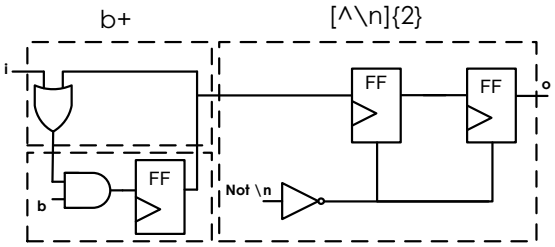


Fig. 7. An implementation for the regular expression $b^{+}[\hat{}\backslash n]\{2\}$.

The above constraint 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. To our advantage however is the fact that more than 95% of the constraint repetitions included in Snort v2.4 and Bleeding Edge IDS regular expressions are of a single character or character class. The rest 5% are repetitions of regular expressions that require multiple and possibly variable number of cycles to match. These cases are implemented via unrolling the constraint repetitions.

Detecting overlapping matches may not be useful when a basic building block forms on its own a regular expression, since in that case the first match is enough to match the regular expression. Then the shift registers of the Exactly and Between block can be reduced to a counter. On the contrary, when a basic block is placed in a larger regular expression, the first match may not lead to the match of the entire regular expression, while another overlapping match may do. There are cases where detecting the last match would be sufficient. For example, in the regular expression $r = a\{3\}bc$, only the last match of $a\{3\}$ block can result in a match of r, (i.e., given an input string $aaaaaabc$). However, detecting only the last match without keeping track of all input tokens is not straightforward.

We describe next an implementation example of the regular expression $b^{+}[\hat{}\backslash n]\{2\}$ illustrated in Figure 7. 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 clock cycle the input 'i' will be high, and the first 'b' will be accepted. Hence, the first flip-flop 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 flip-flops are high and the module outputs a match for the input string "bba". Then, an "$\backslash n$" character

arrives, which resets the exactly block, and therefore a second match for the input string '$ba \setminus n$" will not occur.

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 [22], we perform the following optimizations. As mentioned in the previous sub-section, we employ the SRL16 modules to implement single bit shift registers and store multiple NFA states. Additionally, we share all the common prefixes; that is, 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 FPGA resources for the Snort v2.4 ruleset. Next, each optimization is discussed in more detail.

Xilinx SRL16: Usually, the states of the NFA are stored in flip-flops, each flip-flop representing a single state. An area efficient solution to store multiple states is to configure Xilix LUTs as shift registers (SRL16s). Many basic blocks, such as constraint repetitions, need to store a large number of states, which can also be implemented by shift registers. These shift registers are true FIFOs, and consequently, can be implemented with SRL16s which require a single logic cell to store 17 states (a single LUT plus a flip-flop). 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 some rulesets (e.g., Snort v2.4) a large number of regular expressions have the common prefixes. Consequently, these prefixes can be shared as depicted in Figure 2. 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.

Sharing of Character Classes: 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. As an example, note that there are more than 8,000 character class cases in the Snort 2.4 Oct'06 regular expressions, which are reduced to about 62 unique cases.

Sharing of Static Patterns: Similar to the character classes, this work considers a static pattern matching module to match static patterns included in the regular expression set. We use our previously proposed technique DCAM [23] and share the outputs of the module. 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. The sub-patterns are matched using DCAM because it can be integrated more efficiently with the rest of the Regular Expression Engine compared to other more area efficient solutions such as [24]. As an example, note that the Snort v2.4 Oct'06 regular expressions include more than 2,000 unique static sub-patterns of 35,000 characters in total, and therefore, a large amount of resources is saved.

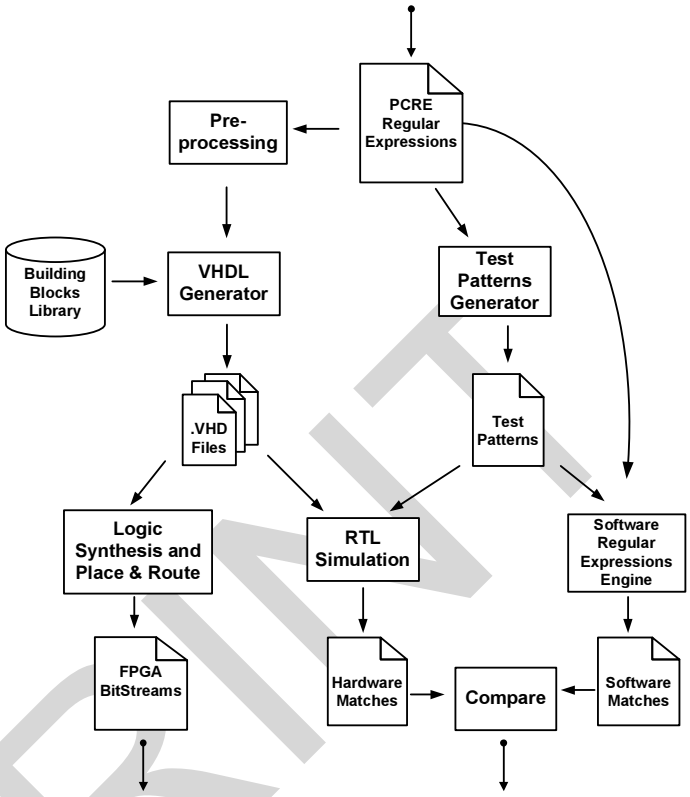


Fig. 8. Proposed methodology for generating regular expressions pattern matching designs.

C. Increase Performance

Two techniques have been employed to improve the performance of the regular expression engines proposed in this paper. The first one keeps the fan-out of certain modules low, while the second one pipelines (when possible) combinational logic. More precisely, like in our previous work [39], this study considers 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. In doing so, the delays of the above connections are reduced at the cost of a few registers. Second, modules such as the decoder, the DCAM and the character class are pipelined. 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 (for device families before Virtex5). The area overhead of this pipeline is zero since each logic cell used for combinational logic includes a flip-flop. Finally, the output of the pipelined modules is correctly aligned with the rest of the design.

V. Synthesis Methodology

In this section we describe the methodology followed to generate regular expression hardware engines from PCRE regular expressions. The methodology is supported by a tool

which generates hardware engines based on the basic blocks previously presented. Figure 8 illustrates the steps used for synthesis and testing of the regular expression hardware engines. Concerning the hardware synthesis of the regular expressions, the tool uses a syntax tree-based approach to generate the structure of the hardware engines. That structure uses building blocks to implement the regular expression primitives. A structural-RTL VHDL code with components described in behavioral-RTL VHDL is generated and logic synthesis, mapping, place and routing are then performed to create the bitstreams able to program the target FPGA.

First, the regular expressions are extracted from the rule-sets. Then, an automatic pre-processing step rewrites regular expressions in order to discard any software related features (conditionals-lookahead) and to change other features (back references) to suit hardware implementation. For example, a conditional-lookahead statement chooses, between multiple regular expressions suffixes, a single one that should be followed, based on the condition. The hardware implementations consider all the multiple suffixes and discard the conditional statement. A back-reference stores the string matched by a sub-RegExp and uses it in a subsequent part of the RegExp. For example, the expression $(a|b|c)\backslash 1$ has a back reference on $(a|b|c)$ which is, e.g., the character a when incoming character a matches the expression $(a|b|c)$. Consequently, the expression $(a|b|c)\backslash 1$ can be matched by the input strings aa, bb, or cc, but not by ab. In our implementation we replace the back-references with the sub-RegExp they refer to (e.g., $(a|b|c)\backslash 1$ becomes $(a|b|c)(a|b|c)$). This way our designs will not miss any matches compared to the PCRE-software implementation, however, may output some extra matches (e.g., $(a|b|c)\backslash 1$ will match the input string ab). A more consistent representation of the back-references is planned for future work. 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 bound to "basic building blocks" which can be automatically mapped to hardwired modules. At this level, the tool can perform a number of optimizations. For example, fully unrolling of certain constraint repetitions (i.e., non single character and non single character classes) is done at this level. Some rules are applied to enable full unrolling of some expressions (e.g., fully unrolling of Between blocks when $\{n, m\}, 0 \leq n \leq 2\ and\ 1 \leq m \leq 3$). These rules are based on the fact that until a certain value of repetitions it is better - area and performance wise - to fully unroll the constraint repetition. The following are examples of rewritten regular expressions. Note that the following rewritten rules are applied for $m > 3$ since for lower values of m the regular expression is fully-unrolled:

$$R\{0, m\} \Rightarrow ((RR?)|R\{3, m\})?$$
$$R\{1, m\} \Rightarrow (RR?|R\{3, m\})$$
$$R\{2, m\} \Rightarrow ((RR)?|R\{3, m\})$$

Performing multiple passes, the tool creates a hierarchical structure of each regular expression in order to generate the VHDL descriptions for the hardware blocks. Figure 9 illustrates an example of a hierarchical decomposition of the regular expression "$\hat{}CEL \setminus s[\hat{}\backslash n]\{100,\}$". First, the tool parses the regular expression, creates the regular expression hierarchy and identifies the basic building blocks (upper part of Figure 9). Then, the parser gathers the information needed for its block. For the example of Figure 9, that is, the characters of the character classes and the repeated expression, and the number of repetitions for the AtLeast block are detected.

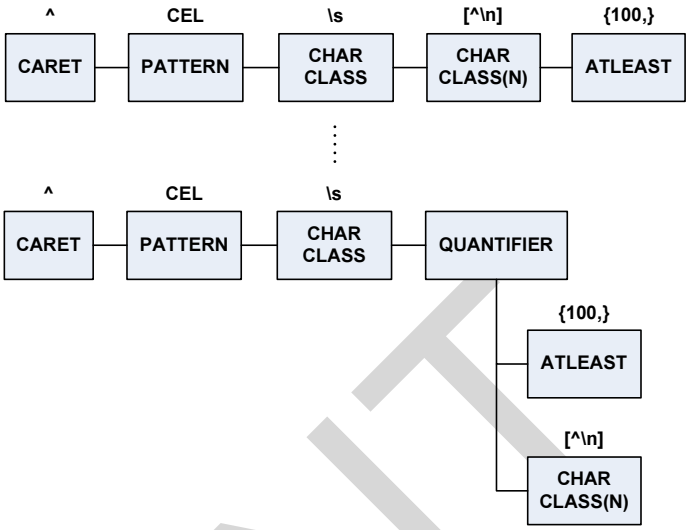


Fig. 9. Example of a hierarchical decomposition.

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

After the VHDL generation, the functionality of the design is automatically tested. Based on the regular expression set, the tool generates input strings covering a subset of possible matches. These input strings are used by the hardware implementations and by a software regular expression implementation. As shown in Figure 8, the hardware implementations are tested by comparing their outputs with the results of the software regular expressions engine.

The compilation of current IDS regular expression sets into VHDL hardware descriptions requires a few tens of seconds, while the logic synthesis, mapping and place & route of the design takes a few hours when the time and area constraints are tight. Looser implementation constraints would lead to shorter execution time. Table IV shows the execution time required in each stage for generating the regular expression hardware engines of Snort and Bleeding rulesets of Oct'06. Snort contains about 5× more regular expressions and therefore requires longer time. The generation of the VHDL code for Snort was completed in 22 seconds, while the synthesis, map and P&R required about 4 hours in total. Compared to Snort, the Bleeding ruleset is substantially smaller. Our tool required 9 seconds to generate the VHDL code, and less than 45 minutes for the subsequent steps. We can observe that the time required for the VHDL generation is negligible compared to the time required for the other stages (from RTL

TABLE IV
GENERATION AND IMPLEMENTATION TIMES FOR SNORT AND BLEEDING RULESETS OF OCT.'06.

Rulesets	# RegExprs	HDL Generation Time (hh:mm:ss)	Synthesis Time (hh:mm:ss)	Map Time (hh:mm:ss)	Place & Route Time (hh:mm:ss)
Snort 2.4 Oct. 2006	1,504	00:00:22	00:57:54	02:24:47	01:30:47
Bleeding Oct. 2006	310	00:00:09	00:01:55	00:26:56	00:16:49

synthesis to the bitstreams ready to be downloaded to an FPGA device). Moreover, the VHDL generation scales better than the subsequent implementation stages as the regular expression set grows. For 5× more regular expressions the compilation time increases only 2.5×, synthesis 29×, and map and P&R about 5.5 ×.

VI. EVALUATION

In this section, we present the evaluation of our regular expression pattern matching designs. The designs have been implemented in Xilinx Virtex2 and Virtex4 devices. The performance is measured in terms of operating frequency and throughput (post place & route results), and FPGA area cost in terms of required LUTs, flip-flops (FFs) and logic cells (LCs). The size and density of the regular expressions sets is evaluated counting their number of non-Meta characters. Meta characters are the ones that have a special meaning/function in the regular expression, the rest are non-Meta characters. Table III presents the number of Non-Meta characters for each basic building block. For example, a character class $[A-Z]$ or a constraint repetition $a\{100\}$ counts as one non-Meta character. This might not be the most indicative metric to measure the size of a regular expression, however, it provides an estimate of the regular expressions sets and enables us to compare against related approaches.

We first present some implementation examples of constraint repetition blocks and evaluate their cost in current FPGAs. Then, we discuss the area reduction and the performance increase achieved with the proposed techniques, by offering a step-by-step optimization flow. Finally, we provide the detailed results of the hardware engines automatically generated using the techniques presented in this paper when all optimizations are enabled.

For evaluation purposes the regular expressions included in three different IDS rulesets are considered. Namely, the Snort v2.4 of April 2006 and October 2006 [17], and Bleeding Edge of October 2006 [18]. Snort v2.4 of April 2006 contains 509 unique regular expressions of 19,580 non-Meta characters in total, while the October version is more than 3× larger having 1,504 regular expressions and 69,127 non-Meta characters. The Bleeding edge ruleset uses relatively fewer regular expressions (310) of 13,441 non-Meta characters in total. Table I includes the main characteristics of these rulesets.

Constraint Repetitions Area Requirements: Figure 10 illustrates the area requirements of the three proposed constraint repetition blocks for different number of repetitions. The exactly block $a\{N\}$ for 10 repetitions (i.e., N =10) needs 5 logic cells (LCs), for N =1,000 it uses 63 LCs, and for

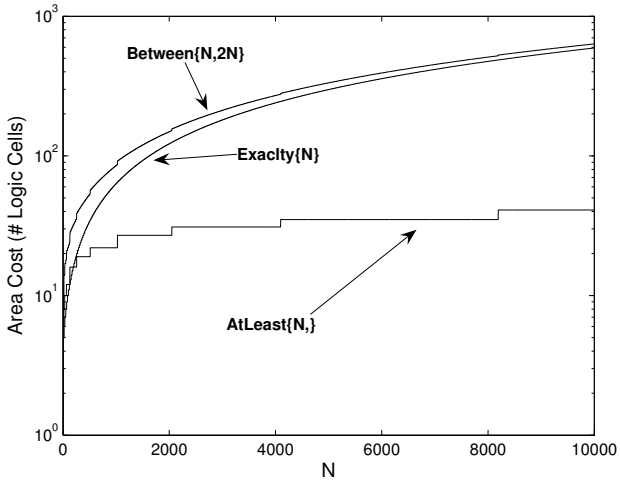
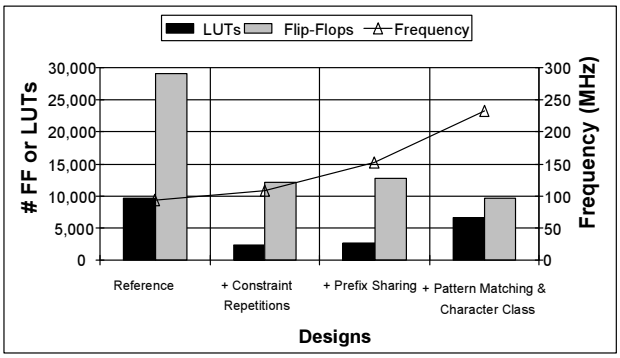


Fig. 10. Area cost of the constraint repetitions blocks.

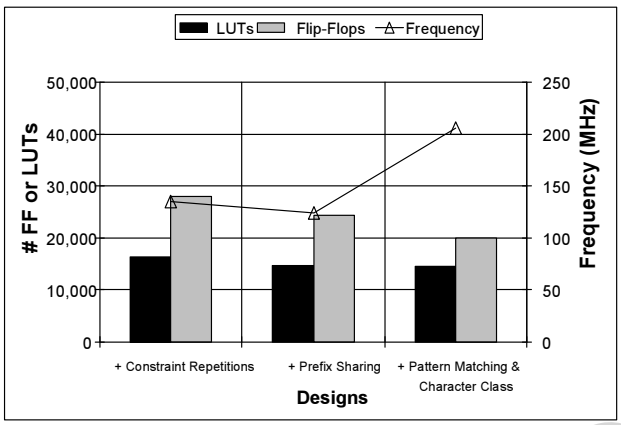
10,000 repetitions needs 593 LCs. Although the Exactly block has $O(N)$ area requirements, the actual cost is only $\frac{N}{17}$ LCs plus a 4-bit counter. The Virtex5 SRL32s would reduce the area cost to $\frac{N}{33}$, while an embedded reset pin in the SRLs would save the 4-bit counter cost. The AtLeast block $a\{N,\}$ scales better as the number of repetitions increases due to its $O(log_2 N)$ area cost. For 1,000 and 10,000 repetitions the AtLeast block needs only 22 and 41 LCs respectively. Finally, a Between block $a\{N,M\}$ of N =1,000 and M =2,000 requires 85 logic cells, and for N =10,000 and M =20,000 needs 634 LCs.

Advantages of our Regular Expressions Optimizations: Next, we show a progressive area and performance improvement applying different optimizations (see Figure 11). The designs have been implemented in a single device (Virtex2-8000-5) in order to perform a fair comparison. The above device is the largest of the Virtex2, however, its speed grade (-5) is lower than other devices of the same family. The lower speed grade and the absence of area constraints is the reason why the results in Figure 11 are slightly different than the best final results depicted next in Table V. For the three sets of regular expressions included in the IDS rulesets mentioned above, three major optimizations are enabled one-by-one. The reference design used to evaluate this proposal is the Sidhu and Prasanna approach [14] combined with the character predecoding technique of [22], [23]. We were able to implement a design for the reference approach only for the Bleeding edge ruleset. In that case, the the number of constraint repetitions is relatively small to fit the design in a single FPGA device. For the rest of the rulesets we only measure the required states needed when unrolling the constraint repetitions operators. The first optimization is to use the constraint repetition blocks previously described in this paper. Subsequently, the prefix sharing optimization is enabled in order to reduce the required area. Finally, the centralized modules which implement the character classes and match the static patterns are included.

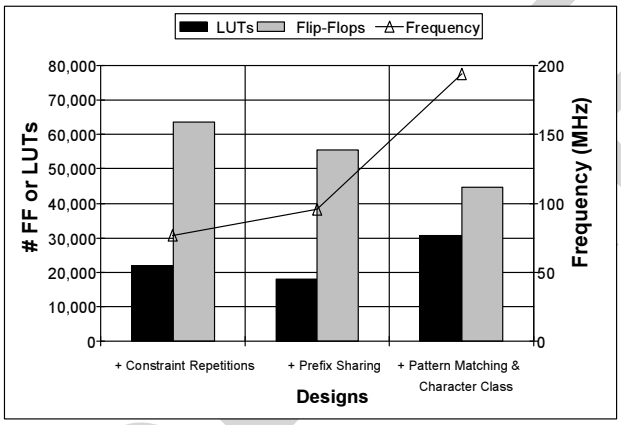
In Bleeding edge IDS ruleset the reference design requires 2.5× more area than the design using the constraint repetition



(a) Bleeding Edge Oct'06



(b) Snort Apr'06



(c) Snort Oct'06

Fig. 11. Area and performance improvements when applying a step-by-step optimization for three different IDS rulesets.

blocks. As depicted in Figure 11(a), that is about 17,000 more flip-flops which correspond to the number of states required when unrolling the constraint repetition expressions. The Exactly and Between blocks store about 15,000 states in about 900 logic cells exploiting SRL16s. Prefix sharing did not reduce the area requirements, due to the small number of regular expressions implemented. When dedicated pattern matching and character classes modules are added then 25% of the area is saved and the maximum clock frequency is improved by 50%. The last design has 3× less area and more than twice the performance compared to the reference one.

Figure 11(b) illustrates the equivalent results for Snort v2.4 of April 2006. This set of regular expressions contains about 700 constraint repetitions that correspond to 470K states when unrolled. Consequently, a reference design would need to store about 470K states more than the one that exploits our constraint repetitions building blocks. Given that about 440K of these states are due to the AtLeast block ($a\{N,\}$) which we implement with an area cost of $O(log_2 N)$, the area savings of the proposed building blocks are increased. We need shift registers only in the Exactly and Between blocks which store about 30K of states in 2,000 logic cells using SRL16. When prefix sharing is applied additionally to the constraint repetition blocks, a 15% area reduction is achieved, while the centralized modules for pattern matching and character classes add another 15% area improvement and a 50% increase in performance. The fully optimized design compared to the one which uses only the constraint repetitions building blocks requires about $1/3$ less FPGA resources and achieves about 50% higher frequency.

Figure 11(c) depicts the area and performance gain when applying the optimizations in the largest regular expressions set used Snort v2.4 of October 2006. The overall number of states required for the ∼750 constraint repetitions when unrolled is about 480K, and 440K of them due to the AtLeast module. In practice, that is the number of extra states required when the constraint repetitions blocks are not used. The 37 Kbits of storage needed for the Exactly and Between blocks are implemented in about 2,200 logic cells. Prefix sharing further reduces area about 15% without significant performance gain. A fully optimized design, using centralized static pattern matching and character classes saves 15% more area and achieves twice the previous maximum operating frequency.

Although the number of required flip-flops is reduced when a new optimization is enabled, this is not the case for the utilized LUTs. Designs that match the static patterns in a separate module require more LUTs than before. Without this optimization static patterns are matched character-by-character as depicted in Figure 4(a). More precisely, the ASCII decoder provides the decoded value of each character, the input token is registered and the inverted decoded character is used for the reset of the flip-flop. This way only a few LUTs are required however a significant amount of flip-flops are used. On the contrary, using a centralized module to match the patterns (DCAM [23]) uses shared SRL16s (each implemented in a LUT) to shift the decoded characters reducing the required flip-flops and increasing the number of LUTs.

In general, our approach results in significant area savings and performance improvements. The dedicated constraint repetition blocks substantially reduce the overall number of states required. The low area requirements of the AtLeast block is especially suitable for IDS regular expressions where the AtLeast statements correspond to over 90% of the number of constraint repetitions states (when constraint repetitions are unrolled). The prefix sharing optimization leads to a further ∼15% area reduction. Moreover, the static pattern matching and character classes modules decrease area another ∼15% and improve the maximum operating frequency by 1.5-2×.

Implementation Results: We further present the detailed

results of the fully optimized designs implemented in the fastest Virtex2 and Virtex4 devices for the three IDS rulesets. The first part of Table V depicts the area cost and the performance results of our designs. More precisely, we report the required LUTs, flip-flops (FFs), logic cells (LCs) and logic cells per matching non-meta character, and the maximum processing throughput for each design. It is noteworthy that all designs process a single byte per clock cycle. Matching the 310 regular expressions of Bleeding Edge ruleset results in about 2.2 and 3.2 Gbps throughput in Virtex2 and Virtex4 devices, respectively. Less than 11,000 logic cells are required which translates to 0.8 logic cells per non-Meta character. The Snort v2.4 ruleset of April 2006 includes over 500 regular expressions and a great number of constraint repetitions. Consequently, it requires 2.5× more logic cells and about 1.28 LCs per non-Meta character. The generated design can support 2 and 2.9 Gbps throughput in Virtex2 and Virtex4 devices, respectively. Although the largest Snort ruleset of Oct'2006 includes 3× more regular expressions, the number of constraint repetitions has increased only 7%. Therefore, the generated design needs only 0.66 logic cells per character and a total of 45,586 logic cells. Note that the overall size of the circuit causes a performance reduction. The maximum throughput achieved is 1.6 Gbps in a Virtex2-4000 and 2.4 Gbps in a Virtex4-60. In general, the number of constraint repetitions in the ruleset and in particular the area consuming ones (Exactly $O(N)$ and Between blocks $O(N + log_2(M - N))$) affect the required resources and the number of LCs per character. For example, both Snort rulesets have similar number of constraint repetitions although the recent one (Oct'06) matches 3× more regular expressions. Hence, the area cost (LC/nMchar) of Snort Oct'06 is substantially better (half) than the one of Snort Apr'06. Finally, and as aforementioned, as the design becomes larger the maximum processing throughput decreases. Snort Oct'06 designs maintain about 75% of the bleeding edge designs performance having a ruleset about 5× larger. Consequently, performance scales relatively well as the ruleset grows, while the area resources per matching character are not significantly affected. Partitioning the designs into smaller blocks similarly to [23], can alleviate performance decrease at the cost however of area overhead. Our preliminary results of partitioned designs show that a 30% performance improvement can be achieved at the cost of 10% increase in resources.

VII. Comparison

Next we attempt a fair comparison with previously reported research on software and hardware regular expression matching approaches.

Recent state of the art software-based solutions offer limited performance and have scalability problems as the regular expression set grows. More precisely, when matching 70-220 regular expressions a NFA approach supports 1-56 Mbps throughput (Yu $et\ al.$ [4]). To provide a faster solution Yu $et\ al.$ propose a DFA solution and rewrite the regular expressions at hand as follows: eliminate closure operands (*, +, ?), e.g., $\backslash s+ \Rightarrow \backslash s$, reduce the repetitions of constraint repetition operators, e.g., $[A-Z]\{j+\} \Rightarrow [A-Z]\{j,k\}$, and do not detect overlapping matches. Hence the accuracy of their implementation is compromised. Their DFA approach requires several Mbytes of memory for only a few tens of regular expressions and achieves 0.6-1.6 Gbps throughput depending on the regular expression set and the input data [4]. Compared to our approach, NFA software approaches support about 40× lower throughput, while DFA software solutions when matching a 10× smaller set achieve 20-65% of our performance.

Next we present a detailed comparison with hardware regular expression matching approaches. Table V contains performance and area results of the most efficient hardware regular expression approaches. In order to compare in terms of area with designs that utilize memory, the memory area cost is measured based on the fact that 12 bytes of memory occupy area similar to a logic cell [40]. 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:

$$PEM = \frac{Performance}{Area\ Cost} = \frac{Throughput}{\frac{Logic\ Cells\ +\ \frac{MEMbytes}{12}}{Non-Meta\ Characters}} \quad (8)$$

Such a metric is commonly used to evaluate the efficiency of FPGA-based static pattern matching designs, e.g., [22], [24], [25], [34]. In the case of regular expressions, the metric differs in the way the non-meta characters are counted. As shown in Table III, we count the Non-Meta characters of a regular expression set as proposed in [20]. We follow a conservative approach, which ignores the number of characters in character classes and the range values in constraint repetitions. Although this approach may hide some of the regular expressions complexity, it enables us to compare against previous works. Finally, the memory requirements of a design should be taken into account. The metric of Sproull $et\ al.$ gives a close estimate of the FPGA area occupied by the memory blocks [40].

Our designs achieve up to 2.5× higher throughput compared to designs that process the same number of incoming bits per cycle and require the lowest area cost. More precisely, compared to Lin $et\ al.$ [32], our design requires the same or up to 2× more resources. Their design needs 0.66 LC per character, while our designs occupy 0.66 to 1.28 LC per character. Unfortunately, Lin $et\ al.$ do not report any performance results focusing only on minimizing the hardware resources and therefore we cannot measure their overall efficiency. Baker $et\ al.$ implemented multiple DFA microcontrollers, which are updated by changing the contents of their memories instead of reconfiguring the FPGA device [35]. Due to this design decision, their module requires about 5-10× more resources than our engines taking into account their memory requirements. Furthermore, they support about half the throughput compared to our solution and have a 10-20× lower efficiency.

TABLE V
COMPARISON BETWEEN OUR REGEXP ENGINES AND OTHER REGULAR EXPRESSION APPROACHES.

Description	RegExp/ Static Patterns[1]	Input bits /cycle	Device	Throughput (Gbps)	Logic Cells[2]	Logic Cells /char	MEM	# Non-Meta chars	PEM
Our RegExp Eng. *BleedingEdge Oct'06*	RegExp	8	Virtex2	2.19	10,698	0.80	0	13,441	2.75
			Virtex4	3.26					4.10
Our RegExp Eng. *Snort Apr'06* [41]			Virtex2	2.00	25,074	1.28	0	19,580	1.56
			Virtex4	2.90					2.27
Our RegExp Eng. *Snort Oct'06*			Virtex2	1.60	45,586	0.66	0	69,127	2.43
			Virtex4	2.42					3.68
Lin *et al.* [32] NFA sharing sub-RegExp	RegExp	8	VirtexE -2000	N/A[3]	13,734	0.66	0	20,914	N/A[3]
Baker *et al.* [35] DFA μ-controllers	RegExp	8	Virtex4 -100	1.4	N/A	2.56	6Mb	16,715	0.22
Sidhu *et al.* [14] NFAs	RegExp	8	Virtex -100	0.46	1,920	66	0	29	0.01
Brodie *et al.* [36] DFAs	RegExp	32	Virtex2	4.0	860	N/A	96Kb	per engine[4]	N/A[4]
			ASIC	16.0	N/A	N/A	27Mb	11,126	N/A[4]
Hutchings *et al.* [20] NFAs	Static Patterns	8	VirtexE -2000	0.4	40,232	2.52	0	16,028	0.16
Clark *et al.* [22] 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.* [21] DFAs	Static Patterns	32	VirtexE -2000	1.18	8,134	19.4	0	420	0.06

Brodie *et al.* implemented DFAs using FSM-based engines aiming at ASIC implementations [36]. Due to their high area cost their entire design cannot be prototyped in current FPGA devices. A single engine of Brodie *et al.* that matches approximately a single regular expression has been prototyped in a Virtex2 device. It achieves 4 Gbps (2× vs. our design), processing 4 bytes per cycle. A single engine requires 860 logic cells and 96 Kbits memory. Their complete design matches 315 Snort-PCRE regular expressions and has a density of 204 chars/mm^2 in a 65 nm technology. Assuming the same technology, we synthesized our largest design in a Virtex5 (65 nm) device. We adjusted only the SRL16s into Virtex5 SRL32s and not our pipeline which is tailored for 4-input LUTs and not the Virtex5 6-input LUTs. Our design matches more than 1,500 regular expressions (69,000 non-meta characters), occupies less than $2/3$ of a Virtex5LX-110 (729 mm^2) which leads to a 142 chars/mm^2 density. Consequently, our approach has comparable area requirements, while we would support roughly 4-5× lower throughput. Despite the lower performance results compared to the above ASIC implementation, there are several advantages to oppose. Brodie *et al.* implementation suffers from the DFA drawbacks such as lack of support to overlapping matches and state explosion. For instance, in case an IDS regular expression when converted to a DFA requires more states than can be stored in the available memory per engine, then this regular expression cannot be implemented. In addition, the implementation and fabrication of an ASIC is substantially more expensive than an FPGA-based solution. Therefore, reconfigurable hardware is an attractive solution for regular expression pattern matching providing higher accuracy, fast time to market and low cost.

Clark *et al.* and Hutchings *et al.* match only static patterns transformed into regular expressions [20], [22] and therefore their designs are simpler. Compared to Hutchings *et al.* we achieve more than 2× their throughput (taking into account that VirtexE devices are about 30-40% slower than Virtex2) and occupy less than half the area. Compared to Clark and Schimmel design that processes 8-bits per cycle, we achieve similar performance requiring 25-50% fewer resources. Our design has similar efficiency (based on the PEM) compared to Clark and Schimmel second design which 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 regular expressions).

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, although, the approach presented in this paper clearly outperforms their designs.

VIII. CONCLUSIONS

In this paper we presented techniques for FPGA-based regular expression pattern matching. More precisely, we described a method to automatically generate hardwired engines that match Perl-compatible regular expressions (PCRE). We introduced three new basic building blocks to implement constraint repetitions and proved that to of them can be simplified without affecting their functionality. Moreover, a number of techniques were employed to minimize the area cost and improve performance. Large regular expressions IDS rulesets were employed to validate the proposed approach. Furthermore, we discussed our methodology and suggested techniques to rewrite PCRE regular expressions in order to suit hardware implementations. Concerning the entire Snort and Bleeding Edge regular expression IDS rulesets, our automatically generated designs achieve a throughput of 1.6-2.2 and 2.4-3.2 Gbps in Virtex2 and Virtex4 devices, respectively. The generated hardware engines require 0.66-1.28 logic cells per non-Meta character. Based on the performance efficiency metric (PEM), our designs are 10-20× more efficient than the best related FPGA approaches. Even compared to designs that match static patterns using regular expressions, and therefore are simpler, our approach has similar and up to 10× better efficiency. In addition, the proposed NFA-based designs have comparable area costs with current ASIC DFA-based

[1]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.

[2]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. The above hold true for device families before Virtex5.

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

[4]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. In a 65 nm technology it is estimated that their module would have a density of 204 characters per mm^2.

approaches. Future work will focus on a more general solution for constraint repetitions, back-references support and more advanced resource sharing techniques.

ACKNOWLEDGMENTS

This work was supported in part by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6). João Bispo would like to thank INESC-ID for the PhD scholarship. João Cardoso would like to acknowledge the support by the Portuguese Foundation for Science and Technology (FCT) - FEDER and POSI programs - under the CHIADO project (POSI/CHS/48018/2002).

REFERENCES

[1] S. Stephens, J. Y. Chen, M. G. Davidson, S. Thomas, and B. M. Trute, "Oracle database 10g: a platform for blast search and regular expression pattern matching in life sciences." in *Nucleic Acids Research*, vol. 33 (database-Issue), Jan. 2005, pp. 675–679.

[2] S. Ray, and M. Craven, "Learning Statistical Models for Annotating Proteins with Function Information using Biomedical Text." in *BMC Bioinformatics.*, vol. 6, Suppl. 1, S:18, May 2005.

[3] J.-M. Champarnaud, F. Coulon, and T. Paranthoen, "Compact and Fast Algorithms for Safe Regular Expression Search," in *Int'l. Journal of Computer Mathematics (IJCM)*, Taylor and Francis Ltd, vol. 81, no. 4, April 2004, pp. 383–401.

[4] 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 *Proc. 2nd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'06)*. ACM Press, 2006, pp. 93–102.

[5] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *ACM SIGCOMM Computer Communication Review*, ACM Press, vol. 36, Issue. 4, Oct. 2006, pp. 339–350.

[6] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-76, May 22 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-76.html

[7] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. of ACM/IEEE Symposium on Architecture for Networking and Sommunications Systems (ANCS'06)*. New York, NY, USA, ACM Press, 2006, pp. 81–92.

[8] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen Polymorphic Processor," in *IEEE Transactions on Computers*, Vol. 53, Issue 11, Nov. 2004, pp. 1363–1375.

[9] K. Compton, and S. Hauck, "Reconfigurable computing: a survey of systems and software." in *ACM Computing Surveys*, ACM Press, vol. 34, no. 2, 2002, pp. 171–210.

[10] G. Berry, and R. Sethi, "From regular expressions to deterministic automata," in *Theoretical Computer Science*, Elsevier, vol. 48, no. 1, Dec. 1986, pp. 117–126.

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

[12] R. W. Floyd and J. D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits," in *Journal of the ACM (JACM)*, vol. 29, no. 3, July 1982, pp. 603–622.

[13] A. Karlin, H. Trickey, and J. Ullman, "Experience with a regular expression compiler," in *Proc. of IEEE Conference on Computer Design/VLSI in Computers*, Oct. 1983, pp. 656–665.

[14] R. Sidhu, and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *Proc. of 9th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, IEEE Computer Society Press, 2001, pp. 227–238.

[15] A. Mukhopadhyay, "Hardware algorithms for non-numeric computation," in *IEEE Transactions on Computers*, vol. C-28, no. 6, June 1979, pp. 384–394.

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

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

[18] Bleeding Edge Threats web site, "http://www.bleedingthreats.net."

[19] 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, USA, 2002.

[20] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting Network Intrusion Detection with Reconfigurable Hardware," in *Proc. of 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, IEEE Computer Society Press, 2002, pp. 111–120.

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

[22] C. R. Clark, and D. E. Schimmel, "Scalable Parallel Pattern-Matching on High-Speed Networks," in *Proc. of 12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, IEEE Computer Society Press, 2004, pp. 249–257.

[23] I. Sourdis, and D. Pnevmatikatos, "Pre-decoded CAMs for Efficient and High-Speed NIDS Pattern Matching," in *Proc. 12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, IEEE Computer Society Press, 2004, pp. 258–267.

[24] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis, "A Reconfigurable Perfect-Hashing Scheme for Packet Inspection," in *Proc. of 15th Int'l Conference on Field Programmable Logic and Applications (FPL'05)*, Tampere, Finland, Aug. 24-26, 2005, pp. 644–647.

[25] G. Papadopoulos, and D. Pnevmatikatos, "Hashing + Memory = Low Cost, Exact Pattern Matching," in *Proc. 15th Int'l Conference on Field Programmable Logic and Applications (FPL'05)*, Tampere, Finland, Aug. 24-26, 2005, pp. 39–44.

[26] M. Roesch, "Snort - lightweight intrusion detection for networks," in *Proc. of 13th USENIX Conference on System Administration*, Seattle, Washington, USA, Nov. 7-12, 1999, pp. 229–238.

[27] M. Rabin, and D. Scott, "Finite automata and their decision problems," in *IBM Journal of Research and Development*, Vol. 3, No. 2, 1959, pp. 114–125.

[28] R. McNaughton, and H. Yamada, "Regular Expressions and State Graphs for Automata," in *IEEE Transactions on Electronic Computers*, EC-9(1), pp. 39–47, 1960.

[29] K. Thompson, "Regular expression search algorithm," in *Communications of the ACM*, vol. 11, Issue. 6, June 1968, pp. 419-422.

[30] M. J. Foster, "Avoiding latch formation in regular expression recognizers," in *IEEE Transactions on Computers*, vol. 38, no. 5, May 1989, pp. 754–756.

[31] C. R. Clark, and D. E. Schimmel, "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," in *Proc. 13th Int'l Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, Sept. 1-3, 2003, pp. 956–959.

[32] C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of regular expression pattern matching circuits on FPGA," in *Proc. of Conference on Design, Automation and Test in Europe (DATE'06)*, Munich, Germany, March 6-10, 2006, pp. 12–17.

[33] J. Moscola, Y. H. Cho, and J. W. Lockwood, "A scalable hybrid regular expression pattern matcher," in *Proc. of 14th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, IEEE Computer Society Press, 2006, pp. 337–338.

[34] Z. K. Baker, and V. K. Prasanna, "A Methodology for Synthesis of Efficient Intrusion Detection systems on FPGAs," in *Proc. 12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, IEEE Computer Society Press, 2004, pp. 135–144.

[35] Z. K. Baker, H.-J. Jung, and V. K. Prasanna, "Regular Expression Software Deceleration For Intrusion Detection Systems," in *Proc. 16th Int'l Conference on Field Programmable Logic and Applications (FPL'06)*, Madrid, Spain, August 28-30, 2006, pp. 418–425.

[36] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching," in *ACM SIGARCH Computer Architecture News*, (also published in *33rd Int'l Symposium on Computer Architecture (ISCA'06)*), Vol. 34 , Issue 2, May 2006, pp. 191–202.

[37] P. Sutton, "Partial Character Decoding for Improved Regular Expression Matching in FPGAs," in *Proc. of IEEE Int'l Conference on Field-Programmable Technology (FPT'04)*, Brisbane, Australia, 6-8 Dec. 2004, pp. 25–32.

[38] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, and S. Vassiliadis, "Packet Pre-filtering for Network Intrusion Detection," in *Proc. 2nd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'06)*, San Jose, California, USA, Dec. 2006, pp. 183–192.

[39] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System," in *Proc. of*

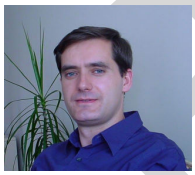
13th Int'l Conference on Field Programmable Logic and Applications (FPL'03), Lisbon Portugal, Sept. 1-3, 2003, pp. 880–889.

[40] T. Sproull, G. Brebner, and C. Neely, "Mutable Codesign For Embedded Protocol Processing," in *Proc. of 15th Int'l Conference on Field Programmable Logic and Applications (FPL'05)*, Tampere, Finland, August 24-26, 2005, pp. 51–56.

[41] J. C. Bispo, I. Sourdis, J. M. Cardoso, and S. Vassiliadis, "Regular Expression Matching for Reconfigurable Packet Inspection," in *Proc. IEEE Int'l Conference on Field Programmable Technology (FPT'06)*, Bangkok, Thailand, Dec. 13-15, 2006, pp. 119–126.

Ioannis Sourdis was born in Corfu, Greece, in 1979. He received his Diploma degree in 2002 and his Masters Degree in 2004 in Electronic and Computer Engineering from Technical University of Crete, Greece. He is currently working towards the Ph.D. in Computer Engineering in the Delft University of Technology, the Netherlands. His research interests include the architecture and design of computer systems, multiprocessor parallel systems, interconnection networks, reconfigurable hardware, network security and networking systems.

João Bispo received a 5-year engineering degree in computer systems and informatics from the University of Algarve, Portugal, in 2006. He is working towards the PhD degree in INESC-ID, Lisbon. In 2006, he spent a period working at the Computer Engineering of the Delft University of Technology, the Netherlands. His research interests include reconfigurable computing, automatic generation of hardware for specific applications, and architecture design exploration.



João M.P. Cardoso received a 5-year engineering degree in electronics and telecommunications from the University of Aveiro, Portugal, in 1993, and the MSc and PhD degrees in electrical and computer engineering from the Instituto Superior Técnico (IST), Technical University of Lisbon (UTL), Portugal, in 1997 and 2001, respectively. He is an Assistant Professor in the Department of Informatics Engineering at the IST/UTL and a senior researcher at the INESC-ID in Lisbon. He was from 1993 to 2006 a faculty member in the Faculty of Sciences and Technology, at the University of Algarve, Portugal. In 2001/2002 he worked for PACT XPP Technologies, Inc., Munich, Germany. There he participated in the research and development of the C compiler for the eXtreme Processing Platform (XPP). He was program chair of ARC'05 and general co-chair of ARC'06, the International Workshop on Applied Reconfigurable Computing. He serves as a Program Committee member for various conferences (IEEE FPT, FPL, ARC, SAMOS, ACM SAC-EMBS, etc.). His research interests include reconfigurable computing, compilation techniques, application specific architectures and design automation of embedded systems. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Stamatis Vassiliadis was born in Manolates, Samos, Greece, in 1951. He is currently a Chair Professor in the Electrical Engineering, Mathematics, and Computer Science (EEMCS) department of Delft University of Technology (TU Delft), The Netherlands. He previously served in the Electrical and Computer Engineering faculties of Cornell University, Ithaca, NY and the State University of New York (S.U.N.Y.), Binghamton, NY. For a decade, he worked with IBM, where he was involved in a number of advanced research and development projects. He received numerous awards for his work, including 24 publication awards, 15 invention awards, and an outstanding innovation award for engineering/scientific hardware design. His 72 USA patents rank him as the top all time IBM inventor. Dr. Vassiliadis received an honorable mention Best Paper award at the ACM/IEEE MICRO25 in 1992 and Best Paper awards in the IEEE CAS (1998, 2001), IEEE ICCD (2001), PDCS (2002) and the best poster award in the IEEE NANO (2005). He is an IEEE and ACM fellow and a member of the Dutch Academy of Science.