Synthesis of Regular Expressions Targeting FPGAs: Current Status and Open Issues

João Bispo[1], Ioannis Sourdis[2], João M. P. Cardoso[1], and Stamatis Vassiliadis[2]

[1] IST/INESC-ID, Lisboa, Portugal, joaobispo@gmail.com,jmpc@acm.org
[2] Computer Engineering, TU Delft, The Netherlands,
{sourdis,stamatis}@ce.et.tudelft.nl

Abstract. This paper presents an overview regarding the synthesis of regular expressions targeting FPGAs. It describes current solutions and a number of open issues. Implementation of regular expressions can be very challenging when performance is critical. Software implementations may not be able to satisfy performance requirements and thus dedicated hardware engines have to be used. In the later case, automatic synthesis tools are of paramount importance to achieve fast prototyping of regular expression engines. As a case study, experimental results are presented, for FPGA implementations of the regular expressions included in the rule-set of a Network Intrusion Detection System (NIDS), Bleeding Edge, obtained using a state-of-the-art synthesis approach.

1 Introduction

Regular expressions can be a heavy computational burden in some applications. For instance, the new generation of Network Intrusion Detection Systems (NIDS) relies heavily in regular expressions, a case where they represent a considerable amount of the total computing time [1]. The set of regular expressions used in those applications grows quickly. Table 1 shows the number of regular expressions in the available Snort [2] [3] and Bleeding Edge [4] rule-sets, all versions from October 2006, with exception of the November 2006 version of Bleeding Edge (last row). It is also shown the number of *constraint repetitions* (i.e., Exactly, AtLeast, and Between quantifiers) presented in the regular expressions of the rule-sets. As can be seen, the rule-sets include a large number of regular expressions and also many *constraint repetitions.* Those numbers are expected to grow since new rules are being continuously added. As an example, the number of regular expressions in the Snort 2.4 version has increased about 2.9× during 2006.

Pattern matching using regular expressions is distinct from static pattern matching, where the input string is matched against other literal strings. In regular expressions, meta-characters with special meaning are used, and a single regular expression can represent several strings. Regular expressions augment the challenges of static pattern matching (e.g., *overlapped matching*) with other ones, such as space explosion (regular expressions can represent very large strings in a very compact form). Hardware solutions for regular expression pattern matching

Table 1. Characteristics of Snort and Bleeding Edge rule-sets with respect to regular expressions

Rules	**Regular Expressions**			
	total	***Constraint Repetitions***		
		Exactly	**AtLeast**	**Between**
Snort 2.4 (Oct. 2006)	1,504	286	319	7
Snort 2.3 (Oct. 2006)	1,500	286	319	7
Snort 2.2 (Oct. 2006)	1,493	258	319	7
Snort 2.1 (Oct. 2006)	1,380	248	318	6
Bleeding Edge (Oct. 2006)	310	58	6	6
Bleeding Edge (Nov. 2006)	317	63	6	6

are already being used in order to achieve high performance demands. Since in most application domains using regular expressions (e.g., data mining, NIDS, email monitoring and inspection, etc.) periodical updates are required, FPGAs seem to be the preferable technology to maintain up-to-date and specialized hardware engines, able to achieve high-performance.

However, synthesis tools to generate the hardware engines from the regular expressions are required frameworks for fast generation of hardware engines. An example of a hardware regular expression engine approach targeted by the synthesis approach presented in [5] is shown in Fig. 1. The design consists of a character decoder (e.g., receives one character each clock cycle and flags the correspondent output) that outputs 256 flags (considering 8-bit ASCII codes) connected to the regular expression engines (one for each regular expression in the rule-set being synthesized). The synthesis tool can also take advantage of the sharing of some hardware blocks (e.g., responsible for prefix shared by more than one regular expression).

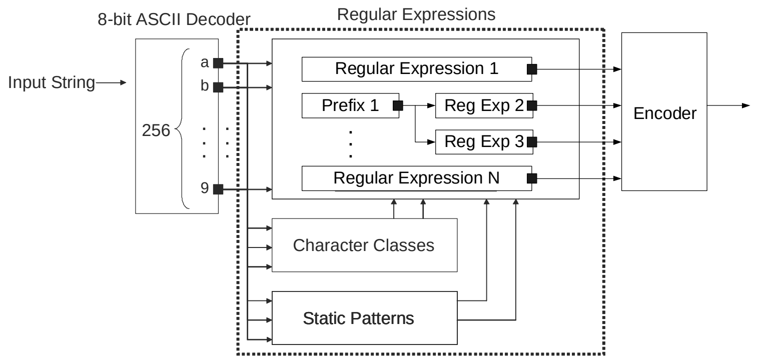

Fig. 1. Block Diagram of the architecture used in [5]

Although several contributions have been made, there are still open issues requiring further research. In addition to a brief explanation of current approaches, a number of those open issues are discussed in this paper.

This paper is organized as follows. Section 2 describes briefly the most relevant approaches to implement hardware regular expression engines. Section 3 discusses a number of open issues. Section 4 shows experimental results related to the implementation of hardware engines for the Bleeding Edge regular expressions. Finally, section 5 draws some conclusions.

2 Implementing Hardware Regular Expression Engines

There are two main approaches to implement regular expressions in hardware: using NFAs (Non-Deterministic Finite State Automats), or using DFAs (Deterministic Finite State Automats). The NFAs have been the solution initially used (see, e.g., the first known approaches to implement regular expressions in hardware [6] [7] [8]), and their inherent parallelism make them appealing for hardware implementations. DFAs are simpler and are the preferable model used in software implementations. Note, however, that DFAs need usually more nodes than NFAs and suffer from state explosion.

Both DFA and NFA based designs have problems when handling some of the regular expressions existent in NIDS (e.g., Snort), mostly because of the large amount of some kinds of quantifiers present (Exactly, AtLeast and Between – referred as *constraint repetitions*). Quantifiers as the previous ones, specifying repetitions of thousands of characters, are common (see Fig. 2 for the Bleeding Edge rule-set), and that is even more prominent in larger rule-sets (e.g., Snort). Most hardware solutions have to represent each of these characters individually (i.e., with full unrolling of repetitions) in order to achieve high-performance demands. From the number of repetitions presented in the new generation of NIDS rule-sets, it can be concluded that full unrolling is not an acceptable solution, because of the large hardware resources required.

NFA-Based Implementations

Regular expressions can be implemented using compound blocks. Sidhu and Prasanna [9] introduced, as far as we know, the NFA based block approach to implement regular expressions in hardware. They introduced the five fundamental blocks: Character, Kleene Star, Concatenation of Characters (Static String) Union and Parenthesis (see Table 2). With these five blocks, any kind of regular expression *defined by a regular language* can be built. Note, however, that designs based on those NFA building blocks implement *constraint repetitions* by unrolling the repeated expression and thus may lead to inefficient hardware engines.

In the paper by Franklin *et al.* [10], the same blocks introduced by Sidhu and Prasanna are used. They use a rule-set of Snort and implement all the static-pattern matching portion with an FPGA. Regular expressions were used as a tool to represent *static strings*, and, to the best of our knowledge, none regular expressions present in the Snort rule-set were implemented.

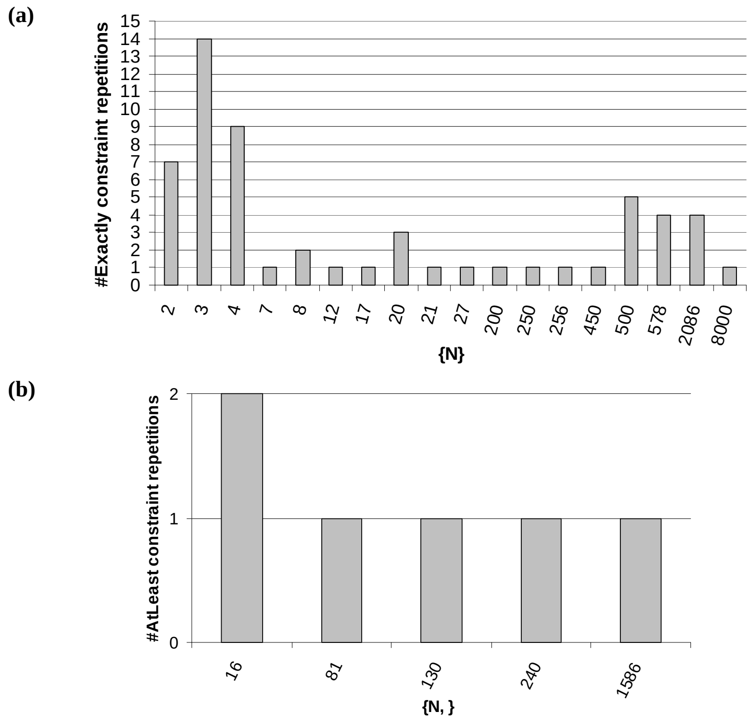

Fig. 2. Distribution of *constraint repetitions* of type: (a) Exactly; (b) AtLeast. Results are for the Bleeding Edge (Oct. 2006 version) presented in Table 1

Table 2. Block primitives employed in the hardware engines as implemented in [5]

(a) Character	(abc...) Static String	(|) Union	(*) Kleene Star
( ) Parenthesis	(ˆ) Caret	($) Dolar	[ ] Character Class
(.) Dot	(?) Question Mark	(+) Plus	
{N} Exactly	{N,} AtLeast	{N,M} Between	

Transition to DFAs

Another NIDS application, this time presenting a complete solution for a "Content-Scanning Module", is presented by Moscola *et al.* [11]. In terms of regular expressions, little is explained in the paper since the focus is on the complete system. To the best of our knowledge, the expressions implemented use the same blocks implemented by Sidhu and Prasanna, plus character classes (there is no mention to *constraint repetitions*). Their work goes a step further, and transform the NFAs extracted from the regular expressions into DFAs, to easily handle context switching (a DFA only has an active state at any time). They also claim that with the DFA approach, the number of states can be reduced most of the time, but the rule-sets used did not include Snort, and it is not explained in the paper what kind of regular expressions are prone to state reduction and state explosion.

The main focus of the work presented by Lin et al. [12] is area reduction, through reusing of common blocks. When implementing regular expressions, there are usually patterns that will be repeated (e.g., "tele" in patterns "telephone" and "television"). They proposed a scheme in order to share the logic of common prefixes, infixes and suffixes. As input, static patterns from an industrial NIDS application and static patterns and PCRE [13] regular expressions from Snort are used. While this is a step forward towards the implementation of more complex regular expressions in hardware, none of those blocks addressed two of the most used features in recent NIDS rule-sets: Character Classes and *Constraint Repetitions*. In addition the new blocks proposed in this scheme are not so compelling in terms of added performance achieved and/or space savings.

DFAs in an ASIC Implementation

Brodie *et al.* [14] presented high-throughput finite state machines (FSMs) for regular expression matching implemented on memory tables rather than logic. This design option was done because they focused on an ASIC implementation, where the memory approach is necessary if regular expressions are to be updated. Their design supports both static patterns and arbitrary regular expressions, and can scan multiple bytes per cycle for achieving high throughput. Although techniques for compression and redundancy minimization are employed, the proposed design architecture is, in terms of resources, prohibitive for FPGA implementations, and compared to the approach used in [5], it consumes much more resources.

DFAs with Microcontrollers

In Baker *et al.* [15], and as with the previous paper, a memory based approach is used. The idea is to make possible to update the regular expressions in the rule-set faster. Using memory, the regular expressions can be trivially updated (e.g., software-like), since the design in the FPGA does not need to be recompiled. The scope is an NIDS application. Also, static patterns and part of Snort regular expressions are both supported. They address the problem of *constraint repetitions*. To prevent state explosion due to the unrolling of certain regular expressions, the wildcards (*, +) and the *constraint repetitions* are handled separately by the microcontrollers, while the simpler patterns are done using DFAs implemented in glue logic. Their approach seems to have the same drawback as the one previously referred: it may require too much overhead when implemented in FPGAs. Basing the approach on microcontroller architectures, they inherit the same problem as the software counterpart – NFA to DFA conversion, because NFA execution is too complex when performed with sequential machines. It is also stated that this approach does not fully support *overlapped matching*.

Extended NFAs

Bispo *et al.* [5] use the same principle from [9] (NFAs in one-hot encoding and accepting one character per cycle), and includes some of the optimizations previously used: the central decoding of [16], and the prefix sharing of [12]. To save area, efficient blocks for *constraint repetitions* were introduced, sharing of other regular expressions components (see the Static Strings and Character Classes blocks in Fig. 1) was used, and the Xilinx SRL16 primitives were employed.

This approach also presented a synthesis methodology to automatically generate the hardware engines. It relies on transforming the regular expressions into a set of block primitives. The primitives used to generate hardware engines are summarized in the Table 2.

Implementations for the blocks referred in the last line of Table 2 have been introduced in [5]. Those implementations deal with *constraint repetitions* of single-cycle blocks (see the AtLeast implementation on Example 1) and implementations of repetitions for multi-cycle blocks without unrolling were identified as an open issue.

Note also that this work has been one of the first to show a fully implementation of a Snort rule-set using FPGAs.

Example 1. Single Character AtLeast Implementation

Fig. 3 shows the single character AtLeast Block proposed in [5]. It flags a match after detecting N or more equal characters. The output remains active until the first mismatch. Doing this, there is no need to store previous states. Even if new signals arrive, indicating subsequent matches, the output will not be affected. The AtLeast block can be implemented using only a counter (up to N) that keeps track of the number of matches.

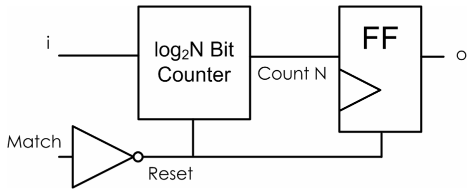

Fig. 3. AtLeast Block

Summary

As a brief summary, Table 3 shows the main characteristics of the previously introduced approaches. Although it is difficult to compare the approaches in terms of the performance achieved, in [5] a metric has been used. However, the metric fully depends on the number of characters in the regular expressions (this has been coined from comparisons with implementations of static strings) and that measure may not be the best one due to the quantifiers present in regular expressions. Unfortunately, a metric, such as the number of characters being scanned per second, is not always possible to be used, because different technologies are usually employed.

Concerning the hardware synthesis of the regular expressions, the tool presented in [5] uses a syntax tree-based approach to generate the structure of the

Table 3. State-of-the-art summary

Authors	**Main Contributions**	**Target: RegExp / Static Patterns**
Sidhu *et al.* [9]	Introduction of NFA block approach.	RegExp
Franklin *et al.* [10]	Pattern Matching using regular expressions. Sharing of common prefixes.	Static Patterns
Lin *et al.* [12]	Sharing of prefixes, infixes and suffixes.	RegExp
Brodie *et al.* [14]	DFAs in memory tables for ASIC implementations. Multiple input bytes per cycle with high throughput.	RegExp
Baker *et al.* [15]	Micro-controllers for *constraint repetitions*. DFAs in FSMs with memory tables.	RegExp
Bispo *et al.* [5]	Extended NFA block approach	RegExpr

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 code is generated and logic synthesis, mapping, place and routing is then performed to create the bitstreams to program the target FPGA.

3 Open Issues

This section presents a number of open issues requiring research efforts in order to improve the hardware synthesis of regular expressions.

Overlapped Matching

One of the known problems is *overlapped matching*. Overlapped matches require new matching evaluations in every position of the input string. Considering as input the string “abc”, and overlapped matches is used, matching tests of “abc”, “bc” and “c” are performed. For a string with N characters, there will be N strings tested. Hence, *overlapped matching* is usually very inefficient in software, since all possible strings need to be tested. An approach such as the one presented by [9], takes advantage of the natural hardware concurrency and executes all *overlapped matching* cases concurrently without penalties. The reason behind this is because at each clock cycle, *every* block in the design sees the same character. In the end of the cycle, the character can be discarded, because all blocks have tested the mentioned character in every possible string position. This is the major reason why it has been difficult to propose a generic block for *constraint repetitions.*

Multi-Character *Constraint Repetitions*

The *constraint repetitions* $e1\{N\}$ (Exactly), $e1\{N,\}$ (AtLeast) and $e1\{N,M\}$ (Between), where *e1* represents a generic regular expression, can be implemented in hardware or software, using NFAs or DFAs. However, their non-unrolled hardware implementation with high throughput, *overlapped matching* and with all the blocks of the engines processing each character in one clock cycle still is an open issue.

For instance, the approach in [5] solves this problem when *constraint repetitions* are applied to regular expressions (*e1*) of the form single Character, single Dot or single Character Class. However, when *e1* is a more complex regular expression, unrolling has been used also as a way to enforce *overlapped matching*. As an example, a possible implementation of a *constraint repetition* evaluating *e1*$\{N\}$ including *overlapped matching* and without requiring a single-cycle implementation is discussed in Example 2.

Example 2. *Constraint Repetitions* with *Overlapped Matching*

Consider the regular expression *(aba)*$\{2\}$. If the input string is “abababa”, with *overlapped matching* it is equivalent to test as input the set “abababa”, “babaaba”, “abaaba”... Thus, a match of the sub-string “abaaba” should be flagged.

Note that when full unrolling the repetitions, all the states are explicitly available and *overlapped matching* is accomplished. However, applying full unrolling to all the *constraint repetitions* may require large amounts of FPGA resources. The approach presented in [5] implements, without full unrolling, repetitions of only one character (it should be noted that most *constraint repetitions* found in NIDS rule-sets are of this kind).

Fig. 4 presents an implementation with *overlapped matching* for the regular expression *(aba)*$\{N\}$. In this case, the 2-bit counter “001” counts sequences of the pattern “001” (the string matching for “aba” only happens after two clock cycles of un-matching) until it reaches N. For every ‘1’ in the input without being in this sequence the counter is initialized to 1 (*overlapped matching*) and it is cleared in the other cases. In this way, *overlapped matching* is accomplished. However, this solution only deals with *e1* expressions (*e1*$\{$N$\}$) matching always the same number of characters per repetition (3 in this example), and a solution for the other cases needs further work.

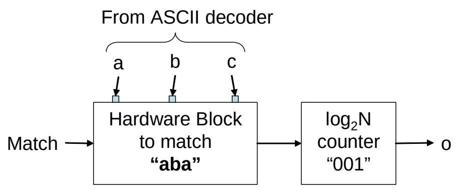

Fig. 4. Block diagram of a possible implementation of the regular expression *(aba)*$\{N\}$ with *overlapped matching* and without unrolling

There is a trade-off between full unrolling and the use of hardware blocks of this kind. The special counters used only make sense for values of N above a certain limit. This is a feature that should be part of a synthesis tool for regular expressions.

Multi-Character Input

There are two ways to improve throughput: higher clock frequency, and higher number of characters scanned per cycle. Higher clock frequency can only be achieved through improved hardware designs or FPGA technology upgrades. Increasing the number of characters scanned on each cycle is, whenever possible, an interesting option to improve performance. However, if *overlapped matching* is supported, that will lead to undesirable area overhead. This happens because all possible offsets must be taken into account [16]. In the paper by [16] it is not clear if it is advantageous to use a multi-character scheme in a one-hot encoding architecture, because when increasing the number of characters scanned per cycle, the number of implemented patterns decreases (due to the needed overhead). This is a problem, because current rule-sets (e.g., Snort) contain thousands of regular expressions. When scanning is done one character per cycle, all characters are position independent (since the blocks only see one character at a time: it does not matter in what position it has arrived). When unrolling is used, all states are explicitly implemented, and it is easier to accomplish *overlapped matching* with multi-character input. Due to the large hardware resources needed, these kind of approaches are now looking for techniques for space-saving [14].

Large Unions

In the NFA approach, one aspect of regular expressions that can have a negative impact in the clock frequency is the Union (same as the logic operator OR). In regular expressions unions of many elements can be used. Also, when sharing common parts of regular expressions union operations are used. With designs working strictly at one character per clock cycle, any union needs to perform its operation during a single clock cycle. The more elements a union has, the slower it may become (when using an LUT-based FPGA). If an union in a regular expression is responsible to a decrease in clock frequency, the synthesis tool should split the regular expression into two or more equivalent ones, using each one a subset of the unions.

Area Reduction

Area reduction in regular expressions is essential. When unrolling all the repetitions in a normal set of Snort regular expressions, up to 500,000 characters are needed to represent in the FPGA (due to the high repetition bounds). The proposed solution was to use special blocks that count the repetitions, instead of representing all the possible states. In repetitions such as AtLeast, a large amount of resources can be saved this way [5]. In repetitions like Exactly, where states are important to the output, they are compactly represented by shift registers implemented with Xilinx SRL16s primitives. Using prefix sharing also saves substantially hardware resources ($\sim$12% less area for a version of the Snort 2.4 rule-set).

There is still work to be done to see if the infix-sufix sharing used by [12] can be supported by the approach presented in [5]. This is currently an open issue.

Hardware Virtualization

Another interesting issue would be the use of dynamic reconfiguration in order to accomplish hardware virtualization. This can be of paramount importance since the NIDS rule-sets are continuously increasing. Hardware virtualization can be exploited by temporal partitioning of the hardware engines executed by time-sharing the target device. At a first glance, temporal partitioning seems not easy to be applied, especially maintaining the one character per cycle processing rate.

Comparing Approaches

A metric often used for area, is the used resources by number of implemented characters. This makes sense for Static Pattern Matching, where all the elements are characters, but not for Regular Expressions. The expressions can be very different in terms of the structures they need (e.g., one expression may rely heavily on *character classes* and other on *constraint repetitions*), and many of the structures they need are independent of the number of characters that appear in the regular expression. A fair metric for regular expressions would be to compare results against fixed sets of regular expressions.

4 Experimental Results

This section illustrates the complexity of the regular expression hardware engines to implement rule-sets of current versions of NIDS. Table 4 shows the results obtained for the hardware engines responsible to implement the rule-set of the Bleeding Edge IDS (with characteristics presented in Fig. 2) with a Xilinx Virtex2 6000, speed grade -6, FPGA. The results were obtained after synthesis, mapping, place and route (Xilinx ISE 8.1 has been used) of the generated VHDL using the synthesis tool previously presented in [5]. The tool receives the regular expressions in the PCRE format and generates the VHDL-RTL (Register Transfer Level) code for the hardware engines.

The VHDL specifications generated consist of about 279,519 and 1,373,095 lines of code (including the character decoder module) for the version using counters and the version using full unrolling for the *constraint repetitions*, respectively.

As can be seen, a large number of FPGA resources is needed to implement the regular expressions of the Bleeding Edge rule-set used in this paper. The number of slices almost doubled when *constraint repetition* are unrolled (third column of Table 4) and surprisingly the maximum clock frequency achieved decreases. This is partially justified by the increase in the number of FFs and routing interconnections, when using unrolling instead of the SRL16 primitives. These preliminary results indicate that using the approach presented in [5], which uses building blocks for *constraint repetition* fully optimized with FPGA primitives, the full unrolling option might not be an interesting design decision.

The maximum clock frequencies of the designs permit to achieve a throughput of 170 and 102 Mega chars per second with the use of counters and with fully

Table 4. Results when implementing the regular expressions of the Bleeding Edge rule-set (October 2006 version)

	non-unrolled (counter-based) *Constraint Repetitions*	fully unrolled *Constraint Repetitions*
Lines of VHDL	279,519	1,373,095
Execution time (RegExpr synthesis + Logic Synthesis + P&R)	∼ 1 hour and 24 min.	∼ 1 hour and 47 min.
#4-input LUTs	2,364	10,761
#FFs	12,533	29,290
#Slices	12,497	24,953
Maximum Frequency (MHz)	170.882	102.543
Maximum Throughput (Mchars/s)	170	102

unrolling, respectively. Note, however, that the experimental results obtained do not take fully advantage of further optimizations that can be exploited such as the use of timing constraints.

Although a large number of FPGA resources are needed, the Bleeding Edge rule-set is relatively small and includes less *Constraint Repetitions* when compared with some heavier Snort rule-sets (see Table 1 for a comparison).

5 Conclusions

This paper introduced current solutions and a number of open issues related to the implementation of regular expressions on FPGAs. A special focus on approaches that can be systematically used to generate dedicated hardware regular expression engines, able to achieve high performance demands, has been addressed. One of the most challenging applications of hardware regular expression engines are the emergent Network Intrusion Detection Systems (NIDS). In order to show the complexity of the hardware engines needed to implement the regular expressions included in current NIDS, this paper includes experimental results for the Bleeding Edge regular expressions rule-set.

Ongoing work focuses on research efforts to circumvent some of the major limitations identified in this paper.

6 Acknowledgments

This work was partially supported 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. Sailesh Kumar, et al., "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *ACM SIGCOMM Computer Communication Review,* Volume 36 , Issue 4, October 2006, pp. 339-350.
2. Snort official web site, http://www.snort.org. Accessed last time on November 2006.
3. Martin Roesch, "Snort: Lightweight intrusion detection for networks," In *Proc. 13th Systems Administration Conference (LISA)*, USENIX Association, November 1999, pp 229–238.
4. Bleeding Edge Threats web site, http://www.bleedingthreats.net. Accessed last time on November 2006.
5. João Bispo, Yiannis Sourdis, João M. P. Cardoso, and Stamatis Vassiliadis, "Regular Expression Matching for Reconfigurable Packet Inspection," in IEEE International Conference on Field Programmable Technology (FPT'06), Dec. 13-15, 2006, Bangkok Thailand, IEEE Computer Society Press, pp. 119-126.
6. 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, pp. 603–622, July 1982.
7. A. R. Karlin, H. W. Trickey, and J. D. Ullman, "Experience With A Regular Expression Compiler," in *Proceedings of the ICCD: VLSI in Computers*, 1983.
8. R. McNaughton and H. Yamada, "Regular Expressions and State Graphs for Automata," in *IEEE Transactions on Electronic Computers*, vol. 9, pp. 39–47, 1960.
9. R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.
10. 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.
11. 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.
12. C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, "Optimization of regular expression pattern matching circuits on FPGA," in *Proceedings of the conference on Design, automation and test in Europe (DATE'06)*, 2006, pp. 12–17.
13. Perl Compatible Regular Expressions website, http://www.pcre.org/. Accessed last time on November 2006.
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 (FPL'06)*, 2006.
16. 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.