Automation Framework for Large-Scale Regular Expression Matching on FPGA*

Thilan Ganegedara, Yi-Hua E. Yang, Viktor K. Prasanna
Dept. of Electrical Engineering, University of Southern California
{ganegeda,yeyang,prasanna}@usc.edu

***Abstract*—We present an extensible *automation framework* for constructing and optimizing large-scale regular expression matching (REM) circuits on FPGA. Paralleling the technique used by software compilers, we divide our framework into two parts: a *frontend* that parses each PCRE-formatted regular expression (regex) into a modular non-deterministic finite automaton (RE-NFA), followed by a *backend* that generates the REM circuit design for a *multi-pipeline architecture*. With such organization, various pattern and circuit level optimizations can be applied to the frontend and backend, respectively. The multi-pipeline architecture utilizes both logic slices and on-chip BRAM for optimized character matching; in addition, it can be configured at compile-time to produce concurrent matching outputs from multiple RE-NFAs. Our framework prototype handles up to 64k "regular" regexes with arbitrary complexity and number of states, limited only by the hardware resources of the target device. Running on a commodity 2.3 GHz PC (AMD Opteron 1356), it takes less than a minute for the framework to convert ~1800 regexes used by the Snort IDS into RTL-level designs with optimized logic and memory usage. Such an automation framework could be invaluable to REM systems to update regex definitions with minimal human intervention.**

***Index Terms*—Regular expression, FPGA, finite state machine, non-deterministic finite automata, NFA, pattern-level optimization, circuit-level optimization**

I. Introduction

Regular expression matching (REM) has traditionally played a key role in text processing and database filtering. More recently, it has become an essential component in the network intrusion detection systems (NIDS) to perform deep packet inspection (DPI). In particular, Perl-Compatible Regular Expression (PCRE) has become a *de facto* REM software library used by many NIDS such as Snort [3] and Bro IDS[1]. For convenience, we call a regular expression written in the PCRE format a *regex*.

In practice, "regular" regexes (which define regular languages)[1] can be matched using either nondeterministic (NFA) or deterministic (DFA) finite automata. The NFA approach [6, 9, 11, 12, 14, 16] is ideal for hardware acceleration using field-programmable gate arrays (FPGA), where a set of regexes are compiled into parallel circuits. Every character position in the regex corresponds to an NFA state, which is set "active" when the character position is reached by the input stream. Numerous optimizations such as input/output pipelining [9], common-prefix extraction [6, 9], multi-character input [14, 16], and centralized character decode [6, 10], can be applied to improve throughput and to reduce resource requirements of the resulting REM circuits.

While various techniques can be used to optimize a particular REM solution, a more daunting challenge is to quickly generate and optimize *any* large-scale REM solution upon regex updates. Such updates can be due to changes in the set of attack signatures used by the NIDS, for example. State-of-the-art designs for hardware-accelerated REM usually require sophisticated optimization procedures that are often tailored to the particular set of regexes. This makes the REM circuit construction and optimization a time and labor consuming task. In contrast, the set of regexes such as NIDS signatures can be updated weekly or even daily. Hence, an automation process to streamline the construction and speedup the optimization of large-scale REM solutions is critically needed.

In this paper, we propose an automation framework which, given a set of regexes, automatically constructs a large-scale REM circuit on FPGA. Improving upon the software toolchain in [16], our framework automates both the parsing of PCRE-formatted regexes (in the *frontend*) and the generation of RTL-level circuit designs (in the *backend*). Based on a modular RE-NFA architecture, the framework can also be extended with custom optimization plug-ins to further minimize resource usage and/or improve throughput performance. Specifically, following are our contributions in this paper:

1) We design an **extensible automation framework** for converting "regular" PCRE regexes into optimized REM circuits in VHDL.
2) We implement an efficient **top-down algorithm** to parse regexes into a modular RE-NFA architecture.
3) We propose a number of **pattern-level and circuit-level optimizations** to reduce resource requirements and to improve memory and throughput performance of the resulting REM solution.
4) Our **multi-pipeline architecture** exploits shared character matching between different regexes and allows a configurable number of concurrent matching outputs.

The rest of this paper is organized as follows. Section II gives the background and related work, while Section III gives an overview of the framework. Section IV and V describe the frontend and backend designs, respectively. Section VI shows performance evaluations. Section VII concludes the paper and discusses future work.

* Supported by U.S. National Science Foundation under grant CCR-0702784.

[1]Some PCRE features such as backreference and recursion are *not* regular. They are not the focus of this work.

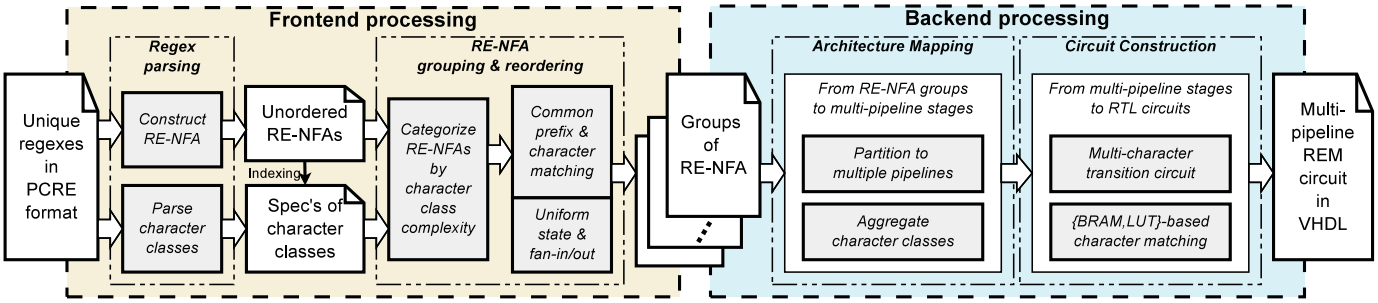

Figure 1. Overview of the frontend (*left*) and backend (*right*) processing flows of the proposed framework. The inner shaded squares are plug-ins which define either a parsing/mapping or an optimization function..

II. Background and Related Work

A. Regular Expression Matching

Any regular expression, by definition, describes a regular language over a fixed alphabet. There are three basic operators which provide the facility to combine individual characters to form arbitrary regular expression patterns: *concatenation* ($\cdot$), *union* ($|$) and *Kleene closure* ($*$). Additionally, most REM software (including PCRE) also support several derivative operators, such as *optionality* ($?$) and *constrained repetition* ($\{\mathtt{a},\mathtt{b}\}$), as well as the pre-defined and custom character classes (see Table I). A regular expression in such a derivative syntax is usually referred to as a *regex*.

Regular expression matching (REM) has been traditionally implemented as software libraries (*e.g.* [2]). Software based REM matches the input stream against each regex by sequentially searching for the matching condition in a depth-first manner. If a search path (following certain choices at various *union* or *closure* operators) fails to produce a valid match, the search is backtracked and started over. Such sequential search and backtracking make software based REM a performance bottleneck in high throughput systems [13]. .

B. NFA-based REM on FPGAs

Hardware based REM implementation was first studied by Floyd and Ullman [8], where an n-character regex is first converted to an n-state nondeterministic finite automaton (NFA), then mapped to an integrated circuit using no more than $O(n)$ circuit area. Sidhu and Prasanna [12] later proposed an algorithm to construct REM circuits on FPGA in a similar NFA architecture, which was also used by most other hardware based REM designs ([6, 9, 11, 14]). Yang and Prasanna [16] adopted a different approach to first translate an arbitrarily structured regular expression of length n to a modular RE-NFA with n modules, then map the RE-NFA to a uniformly structured circuit.

Automatic REM circuit construction on FPGAs was first proposed in [9] using JHDL for both regular expression parsing and circuit generation. In particular, the (J)HDL construction approach used in [9] is in contrast to the *self-configuration* approach done by [12]. Large-scale REM circuit was also considered in [9], where the character input is broadcasted globally to all states in a tree-structured pipeline. In [6], the regular expression was first tokenized and parsed into a hierarchy of basic NFA blocks, then translated into VHDL using a bottom-up scheme. In [11], a set of scripts were used to compile regular expressions into opcodes, to convert opcodes into NFA, and to construct the NFA circuits in VHDL.

A multi-character decoder was proposed in [7] to improve pattern matching throughput. While the technique was claimed to be applicable to REM, only the construction of a fixed-string matching circuit was presented. An algorithm that extends any single-character matching REME *temporally* into a multi-character matching REME was proposed in [14]. In contrast, the modular RE-NFA architecture in [16] allows its circuit to be stacked *spatially* and automatically to process multiple characters per clock cycle.

Although hardware based REM solutions usually outperform software based ones in terms of matching throughput, it is in practice much harder to change the design of a hardware circuit than to update the set of regexes matched by a software program. The problem is aggravated by the numerous sophisticated optimizations applied to the REM hardware designs. Thus an automation framework for constructing and optimizing hardware-accelerated REM is highly needed.

III. Framework Overview

The primary design goal of the framework is to automate the construction and optimization of large-scale regular expression matching (REM) circuits on FPGA in a *configurable* and *extensible* manner. In addition, the framework shall allow various optimizations to be applied effectively and generate high-performance circuits that scale well to large numbers of regular expressions (regexes).

In order to achieve these goals, we follow the example of modern software compiler design to divide the framework into a *frontend*, which handles regex parsing and pattern-level processing, and a *backend*, which constructs the multi-pipeline architecture for REM and performs circuit-level optimizations. Central to this two-phase processing is the modular RE-NFA architecture with which the regexes are represented and manipulated internally by the framework. Figure 1 gives a comprehensive overview of the framework.

The frontend accepts a (potentially large) set of unordered, PCRE formatted "regular" regexes and parse them into a collection of intermediate RE-NFAs, one for each input regex. All the operators listed in Table I are supported by the parsing. The intermediate RE-NFAs are then optimized by the frontend

Table I
PCRE OPERATORS SUPPORTED BY OUR SOFTWARE

Op.	Name	Example	Description
-	Concatenation	q_1q_2	q_2 right after q_1
|	Union	$q_1 \mid q_2$	Either q_1 or q_2
*	Kleene closure	$q*$	q zero or more times
+	Repetition	$q+$	q one or more times
?	Optionality	$q?$	q zero or one times
{*m*, *n*}	Constrained rep.	$q\{m,n\}$	q in m to n times
[...]	Character class	[a − c]	Either a, b or c
[^...]	Inv. char. class	[^\r\n]	Neither \r nor \n

with pattern-level manipulations such as the categorization of RE-NFAs by character class complexity and the grouping of RE-NFAs based on common prefix/character properties. The frontend is also extensible by custom optimization plug-ins as long as the outputs of these plug-ins respect the intermediate RE-NFA representation.

Once the frontend processing is complete, the ordered groups of RE-NFAs are presented to the backend where they are further optimized at the circuit level and mapped to the multi-pipeline architecture on FPGA. The multi-pipeline architecture is capable of matching an input stream of characters against the entire set of regexes and outputting multiple matching results per clock cycle, one per each pipeline. Similar to frontend, the backend can also be extended with custom optimization plug-ins before the optimized RE-NFAs are converted to RTL-level circuit designs in VHDL.

IV. FRONTEND PROCESSING

The frontend is described in two parts: (1) Parsing regex to RE-NFA, and (2) Pattern-level categorization and grouping.

A. Parsing Regex to RE-NFA

To generate the RE-NFA for a given regex, we first extend the *modified McNaughton-Yamada* (MMY) constructions described in [16] to support the additional PCRE operators in Table I. Then we improve the speed performance of the original MMY algorithm by converting regexes to modular RE-NFAs in a tokenized manner.

1) Adding support for additional PCRE operators: In addition to the basic *concatenation*, *union*, and *Kleene closure*, the frontend supports three additional (PCRE) operators: *optionality* (?), *repetition* (+) and *constrained repetition* ({m,n}). Figure 2 illustrates the extended MMY constructions for converting these six operators into modular RE-NFAs. As shown in the figure, both optionality (?) and repetition (+) are special cases of Kleene closure where the feedback and feedforward transitions, respectively, are omitted Depending on whether *m* and *n* are equal to each other or to zero and infinity, respectively, there may be several versions of constructions for the constrained repetition {m,n}. Here we show the general case where we first replicate the repeated sub-regex n times in a chain of sequential transitions, then connect the output from the last n − m + 1 copies of the sub-regex to the following sub-regex with ϵ-transitions.

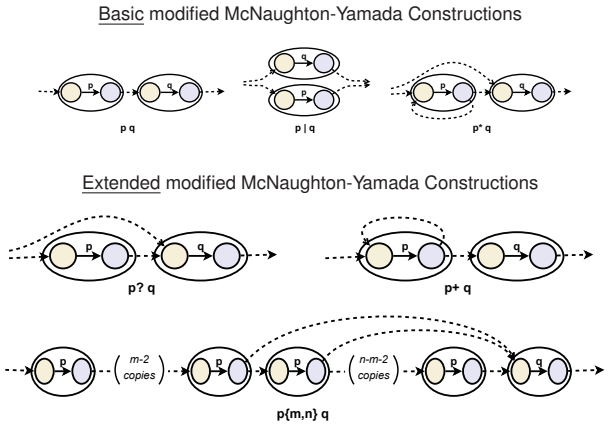

Figure 2. Graphical representation of the basic (*upper*) and extended (*lower*, supporting ?, + and {m,n}) MMY constructions. Each oval represents the a sub-NFA; each dashed line represents an ϵ-transition connecting the output of one sub-NFA to the input of another.

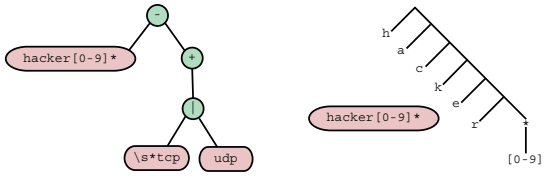

Figure 3. Parsing the regex to RE-NFA: partitioning the regex into sub-regex tokens (*left*) versus parsing a single sub-regex token recursively (*right*).

Recursively, the extended MMY constructions parse the regex until each "oval" in Figure 2 contains only a single character matching. This results in a modular RE-NFA architecture where each oval in the figure can be mapped to a state module in hardware. In addition, the RE-NFA architecture allows character matching (labeled transitions inside the ovals) to be separated from state transitions (ϵ-transitions between ovals), which is critical for mapping to the multi-pipeline architecture in the backend processing (Section V).

2) Tokenized regexes parsing: The original MMY construction algorithm in [16] is highly recursive in nature, which can make the parsing progress inefficient for long regexes (some regexes in Snort rules contains thousands of characters). To speedup the parsing progress, we first partition a given regex into sub-regex "tokens" where each token corresponds to a portion of the regex separated from either a '(', '|' or ')'.

To demonstrate the this concept lucidly, we consider the example of parsing "hacker[0-9]*(\s*tcp|udp)+". The regex is first partitioned into three tokens, "hacker[0-9]*", "\s*tcp" and "udp". Each token can further consist of any of the operators mentioned in Table I. Then, we calculate the entering states and exiting states for each sub-regex token, as summarized in Table II. An entering state is a state through which the matching progress can enter into a given sub-regex. For example, "\s*tcp" has two entering states because of the by-passing transition of "\s*" due to the Kleene closure operator. Similarly, an exiting state is a state through which a sub-regex can exit, which would be the two "p"-matching states for the above sub-regex. Compared to the original MMY algorithm, the tokenized approach significantly reduces the

Table II
ENTERING AND EXITING STATES OF EACH SUB-REGEX AND PARENTHESIS

Sub-regex (token)	Entering states	Exiting states
`hacker[0-9]*`	`h`	`r , [0-9]`
`\s*tcp`	`\s , t`	`p`
`udp`	`u`	`p`
`(\s*tcp|udp)`	`\s , t , u`	`p , p`

depth of recursion and allows tokens in a long regex to be parsed in parallel.

B. Classification of RE-NFAs

We propose two classification techniques to perform pattern-level optimization in that frontend of our framework.

1) Character class complexity: We first classify the RE-NFAs by the complexity of the character matching operations required by the corresponding regex. We define two types of character classes namely, *simple* and *complex*. The simple character classes have one or two characters grouped together while complex type has more than two characters. For instance, `[\r\n]` is a simple character class while `[\r\n\s]` is a complex character class. The same rule applies for the negated character classes (*i.e.* `[^\r\n]` is simple but `[^\r\n\s]` is complex).

The criterion for the above categorization is educed from the specifications of our architecture, which is discussed in detail in Section V-B.

2) Degrees of similarity between regexes: To exploit the benefits of the degree of similarity between regexes, we adopt the method proposed in [5] where, after performing a pattern-level similarity check for all pairs of regexes, a fully connected graph is generated with regexes as nodes and their (pair-wise) degrees of similarity as weighted edges. A graph partitioning algorithm is then performed to group the similar regexes (or more precisely, their corresponding RE-NFAs) together to allow better resource sharing when implementing the RE-NFAs in hardware.

V. BACKEND PROCESSING

Structurally, the multi-pipeline architecture is a two-dimensional array of stages, where each stage consists of 1 to 16 RE-NFA circuits with prioritized matching results. Functionally, the multi-pipeline architecture improves upon the staged pipelining in [16] by offering more flexible matching and optimization capabilities, while preserving the correctness of our previous design:

1) Allow multiple regex matching outputs per clock cycle.
2) Minimize utilization of on-chip block RAM (BRAM).
3) Optimize character matching and state update circuits.

A. Multi-Pipeline Architecture

As shown in Figure 4, the multi-pipeline architecture is parametrized by two values: the number of pipelines (p) and the number of stages per pipeline (k). While all $(p+1)$ pipelines share the same character input, each pipeline has its own matching output and (BRAM-based) character classifier shared by all $(k+1)$ stages in the pipeline.

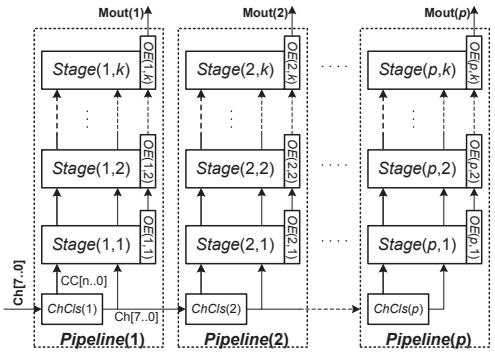

Figure 4. Multi-pipeline architecture ($p+1$ pipelines, each $k+1$ stages).

The multi-pipeline architecture is designed with two philosophy. First, all signals shall be propagated though the entire set of RE-NFAs in a pipelined manner without long routing paths. This can be seen from Figure 4 where both the input characters (`Ch[7..0]`) and their classification results (`CC[n..0]`) are routed locally between adjacent pipelines and stages. Second, the two-dimensional structure of the multi-pipeline architecture shall offer a flexible tradeoff between matching capability and resource usage at compile time. This is further explained in the following subsections where we discuss the effects of multiple concurrent matching outputs verses shared character classifications.

1) Multiple Concurrent Matching Outputs: A critical requirement of large-scale regular expression matching (REM) is to output multiple matching results concurrently. Suchcapability is needed to distinguish the matching results from "conflicting" RE-NFAs at run time. Recall that each RE-NFA defines a regular language over the input characters [4]. We can then define "conflicting" RE-NFAs as follows:

Definition 1: Two RE-NFAs *conflict* with each other *iff* the regular language defined by one RE-NFA intersects that defined by the other RE-NFA, but neither language is a subset (or superset) of the other.

It follows that, with a single matching output, the matching results from two conflicting RE-NFAs cannot be distinguished unless their *intersection* and *difference* RE-NFAs are defined and matched instead. However, defining the intersection and difference of two RE-NFAs is a hard problem and can significantly increase the resource requirement.[2]

On the other hand, with multiple matching outputs, this problem is alleviated as long as the matching results from conflicting RE-NFAs can be output concurrently. To take advantage of this property, we perform a simple two-step algorithm in the backend when partitioning the set of RE-NFAs into multiple pipelines:

1) First we use the available I/O bandwidth to calculate the maximum number of concurrent matching outputs, each generated by one pipeline.

[2]For example, `/[a-z]{16}/` and `/[0-9a-f]{16}/` not only conflict with each other, but their difference RE-NFAs are also very hard to define.

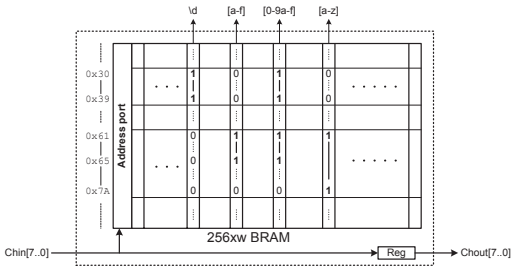

Figure 5. BRAM-based character classifier for w complex character classes.

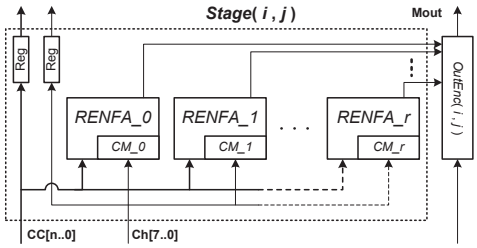

Figure 6. Stage architecture with a priority output encoder.

2) Then we assign RE-NFAs that are known to conflict with each other to different pipelines.[3]

Depending on the particular solution requirement, we can have either a "tall" multi-pipeline architecture with few pipelines and many stages per pipeline, or a "flat" one with many pipelines but few stages per pipeline. In either case, conflicting RE-NFAs can output matching results concurrently and be accurately distinguished.

2) Shared Character Classifications: Each pipeline in the multi-pipeline architecture has a BRAM-based character classifier shared by all RE-NFAs in the pipeline. Figure 5 illustrates an example character classifier where character classes `\d`, `[a-f]`, `[0-9a-f]` and `[a-z]` (among a few unspecified others) are matched in parallel by one BRAM access. In general, BRAM-based character classifier is only used to match *complex* character classes (see Section IV-B) which would otherwise require much circuit logic resource to match.

Since all RE-NFA state transitions with the same (complex) character class can share the output of a single column of BRAM, the number of commom character classes between various RE-NFAs can also be used as a metric for partitioning RE-NFAs into different pipelines. Subject to the I/O constraint, a "flat" multi-pipeline favors more concurrent matching outputs, while a "tall" multi-pipeline favors greater shared character classifications. The height of the multi-pipeline can be configurable at compile time to tradeoff resource efficiency for multi-match capability.

B. Stage Architecture

Figure 6 shows the architecture of a stage with separate character matching circuits (`CM`). Conceptually, all RE-NFAs are separate from one another; practically, the backend can exploit the common prefix and shared character matching among various RE-NFAs (which are grouped together by the frontend based on these properties) to improve resource efficiency. All RE-NFAs in the same stage are prioritized by the output encoder (`OutEnc`) to produce at most one matching output per clock cycle. To maximize flexibility, a stage receives two types of character inputs: (1) a set of character classification results (`CC[n..0]`) propagated from a previous stage; (2) the 8-bit input character (`Ch[7..0]`) generating these classification results.

[3]If the number of mutually conflicting RE-NFAs is greater than the number of pipelines, then some conflicting RE-NFAs must be assigned to the same pipeline and prioritized.

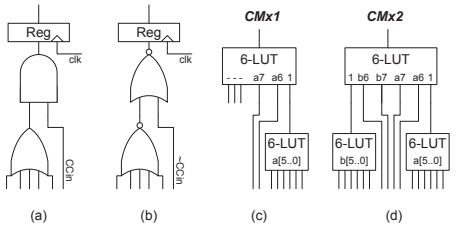

Figure 7. 6-LUT optimized circuit elements: state update modules with (a) normal and (b) inverted character class inputs; compact logic for matching (c) one-value and (d) two-value simple character classes.

While complex character classes are always matched by the per-pipeline character classifier in BRAM, simple character classes can be matched locally in logic as shown in Figure 7c and 7d. Matching characters in logic significantly reduces the utilization of on-chip BRAM, which can be used instead for buffering or other purposes in a larger system. Matching characters locally also helps reducing signal routing complexity, which tends to be high when the number of unique character classes is large.

We adopt the uniform circuit architecture in [16] to implement the RE-NFAs. Specifically, each single character-matching "oval" in Figure 2 is mapped to a state update module in hardware, where the right circle inside the oval corresponds to a 1-bit state register, the left circle corresponds to a fan-in aggregator (an OR gate), and the labeled transition corresponds to a 1-bit character matching (classification) input. In addition, we design two state update modules, one accepting normal character matching (Figure 7a) and the other accepting negated character matching (Figure 7b). This allows the backend to instantiate only one character matching circuit for both a character class and its negation, potentially cut the resource usage of character matching circuits by half.

VI. Experimental Evaluation

A. Experimental Setup

Our framework prototype consists of a C++ program for regex parsing and a number of Bash scripts for the pattern-level optimizations in the frontend; it further consists of a Perl script with various functions for generating optimized mulit-pipeline circuits in the backend. To evaluate the framework prototype, we use the latest Snort ruleset (Feb. 17, 2010) obtained from [3] as our set of regexes. We use Xilinx Integrated Software Environment (ISE) 11.1 to synthesize and place-and-route the multi-pipeline circuit generated by the framework. The target platform for our design is Xilinx Virtex

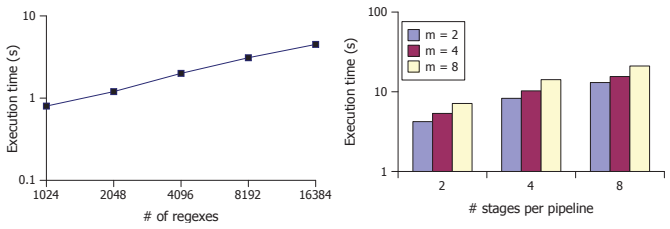

Figure 8. Execution time of the frontend (left) and backend (right) for different sizes of rulesets. The backend construct a 4-pipeline REM circuit matching m = 2, 4 or 8 input characters per clock cycle.

5 family XC5VLX220 with 34k logic slices and 192×1 kb of BRAM. For our experiments, we have used 760 regexes as our test set and show the scalability of our framework for larger rule sets.

B. Regex statistics

We generate several statistics that are useful when implementing the multi-pipeline architecture on our target platform. We discuss two of them here.

Right before frontend processing, we run a duplicate check to remove all the multiple occurrences of a certain rule. The complete Snort ruleset consists of over 20k rules out of which, surprisingly, the number of distinct rules are in the order of 2k. This is an enormous reduction from the logic and memory usage point of view.

The other statistic is the complex and simple character classes of all the regexes. There are 195 different character classes appearing throughout the Snort ruleset and only 108 of them are complex. In otherwords, nearly 45% of the character classes are of simple or negated simple type. Therefore, using the technique described in Section V-B we can have 45% usage reduction of BRAM compared to our previous implementation.

C. Performance Scaling: Frontend and Backend

Figure 8 shows the variation of execution time with different sizes of rulesets and Table III summarizes implementation details for different multi-character matching settings and compares our framework with our previous results in [15] for 2-input character scenario (for a set of 760 REMEs). Our framework prototype can convert thousands of regexes into circuit designs in VHDL in a few tens of seconds. The frontend, written in C++, is roughly an order of magnitude faster than the backend, which was written in Perl. Furthermore, we equip the backend with the plug-in to generate spatially stack REM circuit matching multiple characters per cycle [16]. While improving the matching throughput significantly, we demonstrate that such optimizations can be applied automatically by the framework in only a few seconds.

VII. Conclusion

In this paper, we propose a framework to automate the process of constructing and optimizing a large-scale regular expression matching (REM) engine on FPGA. We divided the framework into two phases, a frontend and a backend, which provided us the opportunity to exploit the possible optimizations in each phase independently of the operations of the other phase. The separation was made possible by our use of the modular RE-NFA architecture to internally represent and handle internally representing the regexes. We developed a tokenized regex parser for the frontend phase and an optimized multi-pipeline circuit generator for the backend phase. Both phases are designed with the ability to be further extended by the user with custom plug-ins.

Table III
FPGA resource usage and clock rate for different multi-character matching (m) settings for 760 REMEs

m	LUTs	BRAM	Clock Rate (MHz)	Compilation Time (min)
2[15]	31 k	216 Kb	303.2	-
2	30 k	69 Kb	276.3	41
4	47 k	138 Kb	202.9	54
8	84 k	345 Kb	178.3	112

References

[1] Bro Intrusion Detection System. http://bro-ids.org/.

[2] Perl Compatible Regular Expression. http://www.pcre.org/.

[3] Snort network instrusion detection. http://www.snort.org.

[4] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., 1972.

[5] Zachary K. Baker and Viktor K. Prasanna. A Methodology for Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *IEEE Sym. on Field Programmable Custom Computing Machines*, April 2004.

[6] João Bispo, Ioannis Sourdis, João M. P. Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *Proc. of IEEE International Conference on Field Programmable Technology (FPT)*, pages 119–126, December 2006.

[7] C.R. Clark and D.E. Schimmel. Scalable pattern matching for high speed networks. In *Proc. of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 249–257, April 2004.

[8] Robert W. Floyd and Jeffrey D. Ullman. The Compilation of Regular Expressions into Integrated Circuits. *Journal of ACM*, 29(3):603–622, 1982.

[9] B. L. Hutchings, R. Franklin, and D. Carver. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proc. of 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, page 111, 2002.

[10] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of Regular Expression Pattern Matching Circuits on FPGA. In *Proc. of Conference on Design, Automation and Test in Europe (DATE)*, pages 12–17, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[11] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proc. of 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems (ANCS)*, pages 127–136, New York, NY, USA, 2007.

[12] R. Sidhu and V.K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proc. of 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, 2001.

[13] R. Smit, C. Estan, and S. Jha. Backtracking Algorithmic Complexity Attacks against a NIDS. In *Proc. of 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 89–98, Dec. 2006.

[14] Norio Yamagaki, Reetinder Sidhu, and Satoshi Kamiya. High-Speed Regular Expression Matching Engine Using Multi-Character NFA. In *Proc. of International Conference on Field Programmable Logic and Applications (FPL)*, pages 697–701, Aug. 2008.

[15] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In *Proc. of 2008 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, November 2008.

[16] Yi-Hua E. Yang and Viktor K. Prasanna. Software Toolchain for Large-Scale RE-NFA Construction on FPGA. *Intl. Journal of Reconfigurable Computing*, 2009:10, 2009.