IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)
Volume 6, Issue 4, Ver. II (Jul. - Aug. 2016), PP 50-58
e-ISSN: 2319 – 4200, p-ISSN No. : 2319 – 4197
www.iosrjournals.org

A Highly Compressible Regular Expression Matching Circuit for Network Intrusion Detection Systems: An ECD-NFA approach

Bala Modi[1], Gerald Tripp[2]
[1] *(Department of Mathematics, Faculty of Science, Gombe State University, Gombe Nigeria)*
[2] *(School of Computing, Faculty of Science, University of Kent, CT2 7NF, Canterbury, United Kingdom)*

Abstract: *Network attacks that flow through network firewalls or network intrusion detection systems (NIDS) are often identifiable by the patterns of data that they contain. The patterns are normally represented by complex regular expressions which are matched at a very high speed. The regular expressions are built into their equivalent automata, using minimal hardware resources in order to detect variations of these patterns. This paper explains the design, structure, and suitability of a hardware-based automata implementation. The approach is based on an input compression technique that uses Equivalence Classification (EC) technique. The technique is used to drive a novel Nondeterministic Finite Automata (NFA) referred to as Equivalence Class Descriptor NFA (ECD-NFA). The ECD-NFA approach creates classes of compressed inputs represented by positive integer values simply referred to as ECDs. The ECDs are class descriptors, which are used as inputs to drive the automata, instead of unclassified raw character-input strings. The ECD-NFA design is built to take advantage of the parallelism provided by Field Programmable Gate Array (FPGA) technology. The design further exploited the FPGA to provide high throughput and support for quick updates. The ECD-NFA design clocks at 460.00 MHz, with a throughput value of 3.68 Gigabits (Gbps). The design incurs very minimal logic circuit cost, and the preliminary results look promising.*
Keywords: *ECDs, ECD-NFA, FPGA, LUTs, Throughput.*

I. Introduction

Network security or policing has become a serious concern for the increasing number of computer network users and Internet Service Providers (ISPs) across the world. Most network security systems usually rely on *layers* of protection and are made up of multiple components including network monitoring and software protection. In addition to hardware and other related appliances, all the network system components work in unison to increase the overall security of a computer network.

As network bandwidths have continued to increase, so have the frequency of network attacks and illegal accesses. These frequent attacks are capable of compromising even the most well secured networks. Network attacks have severe consequences to the privacy and confidentiality of both network clients and confidential documents. These attack patterns could be in form *spam, bugs, denial-of-service (DoS)*, and *malicious software* such as: *viruses, worms, Trojan horse, spyware, hybrid, droppers* and *blended threats* [1]. As such pattern matching has become necessary for finding predefined patterns in a wide range of data streams [2]. A pattern set can be composed of thousands of patterns and could keep growing in order to enforce new policies related to security issues [2]. The matching processes could be considered to be regular expression or exact string matching.

Exact string matching on a given packet can be performed during the process of deep packet inspection of the packet payload flowing into a given network. However, the problem with string matching is that it has become inadequate due to the complex nature of the current patterns of network attacks. To deal with the problem attributed to exact string matching, most popular and current software tools [3], [4], [5] now use *regular expressions* or simply termed 'regexp' to describe payload patterns [6]. A regexp is a regular language constructed with character classes that is defined over a fixed alphabet. A regular language has three basic operations performed on its character classes [7] namely: *concatenation* (.), *union* (|) and *Kleene closure* (*). By properly compounding the three basic operators mentioned, more complex regexps can be constructed. Yang et.al [7] also added that some other common operators such as: *optionality* (?) and *quantified* repetitions like ({*a,* }, {, b} and {*a,* b}), could be constructed by using combinations of the three mentioned basic operators.

Regexps remain more expressive compared to simple patterns of exact–match strings [6], but their implementation process requires huge memory space and bandwidth. Current software solutions for regexp pattern matching have become inadequate in coping with the speed of the current frequency of network attacks. Currently, network speeds are measured in several Gigabits per second, prompting the need for alternative solutions. The alternative solutions are hardware-based regexps pattern matching designs, which are based on hardware technologies such as the: Application-specific Integrated Circuits (ASICs) [8], Graphic Processing Units (GPUs) [9], [10], and Field-programmable Gate Arrays (FPGAs) [11], [12], [13], and [14].

According to Yang et al. [7] 'any given pattern that can be matched by a regexp can also be matched by an automaton', and the preferred choice of automata includes: Deterministic Finite Automata (DFA) or Nondeterministic Finite Automata (NFA). A variation of the former and the latter automata is called the *hybrid* DFA-NFA (hybrid-FA) [6]. Becchi and Crowley [6] explained that DFAs have a 'predictable memory bandwidth', having a deterministic number of memory accesses. The DFA can process an input string using a DFA state traversal per character [6]. The NFA [15] on the other hand 'requires that the number of states… [needed] to represent a regexp should be in the order of the number of characters present in the regexp itself' [6]. The combination of the advantages of both the NFA and DFA lead to the creation of a hybrid-FA [6]. On a typical software implementation, regexps are first converted to either NFAs or DFAs. A DFA processes each character in constant time O(1), but requires $O(2^n)$ memory [16], which can degrade performance [17]. With NFAs, each character is processed in $O(n)$ time, and requires $O(n)$ memory.

However, the NFA processing time could be reduced to O(1), but require $O(n^2)$ memory [15] on FPGAs, which is achieved by exploiting its fine-grained parallelism. Parallelism is the fundamental advantage of FPGAs over microprocessors for regexp matching. As a result, NFA-based approaches are increasingly being appreciated, because they are suitable for exploiting the parallelism and *re-configurability* provided by current FPGAs. However, the use of DFAs for representing regexps that appear in most current rulesets may not be adequate. This is especially true when dealing with regexps composed of multiple wildcards (.*) that are repeated a number times. Such regexps are capable of generating a DFA with potentially millions of states, leading to a condition known as *state explosion*. A good example is the regexp having: *prefix.{100} suffix* [6], where both the prefix and suffix are some regexps. The NFA on the other hand also suffers from the problem of large memory bandwidth requirement, unless some clever minimization or compression strategy is applied to it to reduce the bandwidth requirement. The ECD-NFA approach in this paper is designed to reduce the huge memory bandwidth requirement attributed to most naïve NFA approaches.

The ECD-NFA design uses *equivalence classification* technique to classify all the input strings that have the same effect on the automata. The technique creates the relevant ECDs or class descriptors. The ECDs are then assigned in ascending order, before using them to drive the automata. The matching process utilizes Block Random Access Memories (BRAMs), which are used to compress all the raw data inputs into their various equivalence class vectors of next state transitions. The equivalence classes are then mapped to their respective ECDs accordingly. The BRAMs are later synthesized on a target FPGA device, during the pre-synthesis stage of the design hardware phase as described in Section III.D. The contents of the BRAMs are then fetched and used to compare against the input strings that are streamed into the ECD-NFA for matching to occur within the associated NFA matching units. The description and evaluation of the ECD-NFA structure is discussed by comparing the preliminary results obtained to the other known NFA-based designs. The regexps utilized in this design are drawn from rulesets found in the popular Snort community rulesets [3]. The main contribution of the ECD-NFA is that this type of classification strategy is unique and only suitable for NFAs rather than DFAs, and that the matching of input strings is performed at a very high throughput. The matching process also incurs minimal logic circuit cost in comparison to the other related approaches. The ECD-NFA design is also suitable for implementation on any high speed network that is capable of performing exact pattern matching.

The remainder of this paper is organised thus: A brief summary of related works is described in Section II. Section III describes the algorithm used for constructing the ECD-NFA automaton, and also describes the process used in creating ECDs. Section III also explains the overall structure of the ECD-NFA design and the preliminary evaluation of the results obtained for our ECD-NFA design. Section IV compares the various related designs under consideration by using charts to evaluate preliminary results obtained from Section III. The preliminary results obtained for the related designs are implemented using FPGA implementation tools. The combined results are discussed in order to give a concise view of how each design performs against the other related designs. Finally Section V discusses the conclusion and ideas for future work.

II. Related Works

The related works discussed in this paper are Finite State Machine (FSM) based approaches that utilize FPGA technology. The NFA logic described in [15] produces an output in form of a binary tree. A placed and routed netlist was built before generating *configuration bits* (bitstreams) at runtime in [17]. The bitstream file is what is needed to finally program the FPGA device. Sidhu and Prasanna [15] while implementing NFAs as logic, realized that if all the source input Flip-Flops (FFs) to the destination input FFs are on ϵ–transitions (epsilon transition), then the FFs can be eliminated without being implemented at all, and that could reduce the overall logic circuit size.

The design described in [15] has set the pace for several other approaches that consider building reconfigurable [18][12]. A Java-based design tool called Java-based Hardware Description Language (JHDL) [19] was proposed. The JHDL tool is capable of extracting regexps from the popular Snort rulesets database [20] and then generates the needed Electronic Data Interchange Format (EDIF) files [21] which is replaced by

the Native Generic Circuit (NGC) file. The NGC file contains both the logical design data and the associated design constraints file. The file then is processed by the Xilinx ISE Proprietary Project Navigator application version P.49d, volume 14.4 (nt64) Software. The software is bundled with the Xilinx Synthesis Tool (XST) place and route (PAR) software and is used to produce the required FPGA bitstreams. The bitstreams are used to configure the target Virtex-6 FPGA device. Hutchings, Franklin and Carver [18] extended the work in [15] to include additional functionality for the metacharacter: optionality (?). The operation for the optionality [18] is used to match zero or one input character per clock cycle, which enables faster matching. Tripp [22] proposed an FSM design that operates on a single byte wide data input, while providing a separate FSM for each byte wide data path from a multi-byte input data word. Combining the outputs from the separate FSMs in a given way ensures that string matching is performed across multiple FSMs per clock cycle. A design was also implemented in [23] that resolved the problem attributed to directly sharing infix and postfix sub-patterns. Infix and postfix sub-patterns share similar matching characters that occur either at the start for infix or end for postfix that occur in a given regexp pattern. The design in [23] memorizes the path that the trigger signal emanates from, based on specific constraints suitable for both *exact* string matching and complex regexp matching.

A Perl Compatible Regular Expression (PCRE) compiler that converts regexps from the Snort ruleset into PCRE *opcodes* was implemented in [24]. The opcodes are instructions for the software based PCRE engine defined in a file called *pcre_internal.h* which is part of the PCRE Package. The compiler translated the PCRE opcodes into VHSIC Hardware Description Language (VHDL) codes necessary for parallel implementation in the FPGA. Mitra et al. [24] in their design used a wider input bus through an SRAM interface, which helped to increase the overall matching throughput. An automatic architectural optimisation approach was implemented by Yang et al. [7] which spatially stack regexps matching circuits called REMs to form multiple character matching (MCMs) circuits. The MCMs are then grouped into clusters and marshaled onto a two dimensional staged and pipelined structure. The structure is aimed at improving the overall design clock rate.

However, the problem with the architecture designed in [7] is that the process of distributing and buffering the character matching signals was error prone and difficult to implement when done manually. To address the problem, the approach proposed in [25] used a heuristic that automatically marshaled *k*-REMs with total *N*-states into *p*-pipelines. The process calls a function that compares every character class within each REM with those previously collected in the BRAM, whenever a REM is to be added to an existing pipeline. The matching outputs [25] of each of the REMs are prioritized. The REM with higher priority is given to the lower-indexed pipelines and stages for the sake of efficiency.

The design in this paper combines many of the strategies reviewed into a single design. Such strategies include: character classification and creation of REMs to form MCMs. The concept of classification of character input strings for driving the ECD-NFA-based automata at a more appreciable clock rate is what the approach is mostly concerned with. The ECD-NFA-based design is targeted at utilizing and exploiting the advantage of the parallelism provided by the current FPGA technology. The designs in [26], [27], [28] and [22] utilized a similar concept but implemented it for a DFA approach instead.

The ECD-NFA design is constructed into blocks of MCMs, aimed at implementing a classification based approach that works with NFAs. The design is then staged and pipelined in a stacked formation. The formation helps perform a matching process that favorably competes with the rival approaches. The ECD-NFA design much like the one in [28] is able to take on a given number of regexps and generate the equivalent VHDL codes, ready for logic synthesis. The automatic generation of VHDL codes is paramount to the ease of re-configurability of ECD-NFA design. This is because changes could easily be carried out speedily in the software parser more accurately than in the hardware design environment. Moreover, the delay attributed to the process of re-syntheses before the PAR process of the whole design is greatly reduced on the target FPGA platform.

III. The ECD-NFA Classification Algorithm

The concept of equivalence classification can best be understood based on the definitions discussed in [29] and as discussed herewith.

A. Definitions

i. ***Relation:*** A relation ($\simeq$) on a set S is a subset of $S \times S$. Given any two elements a, b $\in S$, we have the relation a $\approx$ b, which shows that 'a' is related to 'b'. It then follows that if '$\approx$' is a relation of the given set X, then '$\approx$' can be said to be:

a. Reflexive, given that $\forall\, a \in X$, $a \approx a$.

b. Symmetric, given that $\forall\, a, b \in X$, if $a \approx b$, then $b \approx a$.

c. Transitive, given that $\forall\, a, b, c \in X$, if $a \approx b$ and $b \approx c$, then $a \approx c$.

ii. ***Equivalence relation*** **:** A relation $\simeq$ on the set S is said to be an equivalence relation if all three conditions in III.i holds.

iii. ***Equivalence class:*** Given that $\simeq$ is an equivalence relation on a set *S*. Then $a \in S$, an equivalence class of *a* is represented by $[a]$ to be the set: $[a] = \{x \in S \mid x \simeq a\}$.

Algorithm 1: Construction of an *n-ECD* vectors of next states

---

INPUT: An *n-state m-character class* ECDs. The input state is $s_.$
OUTPUT: An *n-state ECD-NFA* with the associated *multi-byte* table of *compressed* ECDs.
BEGIN

i. Read and parse the regexps to be constructed into the equivalent ECD_RTS-NFA.
ii. For $\forall\ i < n$, where $i = 0,1,2,3....n\text{-}1$, and n is the total number of states in the NFA. If the transition (link) from state s_i is a self-transition from state s_i to itself upon consuming a non-empty character, remove all such self-transitions.
iii. For $\forall\ i < n$, $j < n$ and $k < n$, if the output of state s_i connects to the state inputs of some state s_j upon consuming an empty string (ϵ), remove all such transitions $t_{i,j}$ linking state s_i to s_j. Create a new transition that connects state s_i to states s_j and s_k where $s_k > s_j$ on a non-empty input.
iv. For $\forall\ i < n$, $j < n$ and for each transition $t_{i,j}$ from a state s_i to a state s_j, scan through. Store all next states transited to on the same input, into a set of next states. Store all the different sets of next states into a single vector of sets of next states and assign a single input character class descriptor to them.
v. Assign to each classified inputs created in (iv) ECDs, which are the class descriptors. The ECDs now represent the sets of vectors of next states for all character classes that trigger transitions from a state s_i to a state $s_{j,}$ where $i < n$, $j < n$.
vi. Repeat steps (i) - (v) for $\forall\ s_i$, $i < n$ and store all the sets of vectors of next states in a list of state vectors for $\forall$ states s_i in the ECD_RTS-NFA.
vii. Once step (vi) is completed, the process of building the compressed table of ECDs begins. The process first performs the cross product computation of any two sets of vectors of next states v_i and v_j $\forall\ i < n$, $j < n$ contained within the list of state vectors stored in (vi). Subsequently, all the similar vectors are merged to become a single vector. Recursively performing step (vi) – (vii) generates a 4-byte table of ECDs two 2-byte tables.
viii. Finally, exit the process after step (vii) and generate the VHDL file for the ECD_RTS-NFA. The file is then uploaded to the XST VHDL synthesis tool for synthesis and implementation.

END

B. Creating the Vector of Next States for ECDs

Once the design compiler extracts the regexps from the Snort rulesets, the parser then constructs the ECD-NFA. It then goes through the ECD-NFA and begins to create the equivalent classes of all the inputs (ECDs) that have similar effect on the automata. The parser first creates a vector of next states for each current state and then assigns ECDs to each set of next states found throughout the entire NFA. In order to properly explain how Algorithm 1 creates its ECDs when converting a regexp into an ECD-NFA, let us consider the regexp: "*/(a|b)*(cd)/"*.

The various column vectors of next states represent transitions from a given current state to all the various next states to which a given class of inputs have the same effect on the automata. The top rows of Figure 1 are the various classes of inputs used to represent the sets of vectors of next states on the ECD-NFA, while the columns are the vectors of next states transited to from each given current state to the various next states. We present an illustration of the sets of vectors of next states generated by Algorithm 1 when implemented for the regexp "*/(a|b)*(cd)/"*.

Figure 1 shows the various sets of vectors of next states that are transited to for each current state 0-4. The classes of inputs have designated ECDs associated to each one. The NFA with their transitions labeled with ECDs is as shown in 0, while the NFA with transition labeled with sets of input classes is shown in 0. It then follows that a string of inputs such as: "*caaacd*" will be rejected by the ECD-NFA, while a string such as "*aacd*" or "*bacd*" will be accepted by the ECD-NFA as a matching string.

ECDs:	0	1	2	3
Class of inputs:	a,b	c	d	z
state 0:	012	012	012	012
state 1:	12	-	-	-
state 2:	-	3	-	-
state 3:	-	-	4	-
state 4:	-	-	-	-

Figure 1: ECD-NFA transitions based on each input of the various ECDs. The letter *z* represent all those characters in the ASCII character set that are not *a,b,c,* or *d* (or simply *[^abcd]*.

C. The ECD-NFA Construction Process

Based on the regexp "*/(a|b)*(cd)/"* and the ECDs used to represent the various vectors of next states as seen in Section III.A, the NFA shown in 0 is first constructed before minimizing it using a simpler process to create the more compact ECD-NFA as shown in 0 and seen in Algorithm 1. 0 shows the constructed ECD-NFA with ECDs assigned as the new inputs for the transitions.

It is interesting to know that 0-4 all perform the same matching process, but in different reduced forms. The total number of states and transitions on the original NFA as seen in 0 have now been reduced from 11 states to a mere 5 states as seen in 0-4. The number of transitions has also reduced from 13 transitions to as little as 7 transitions. This reduction shows that while minimizing an original non-ECD-NFA to the ECD-NFA, about 45% of the original NFA states have been eliminated, while about 53% of the transitions in the original NFA have also been eliminated. This creates a more compressed and compact ECD-NFA driven by ECDs rather than the unclassified raw characters of the incoming strings of inputs.

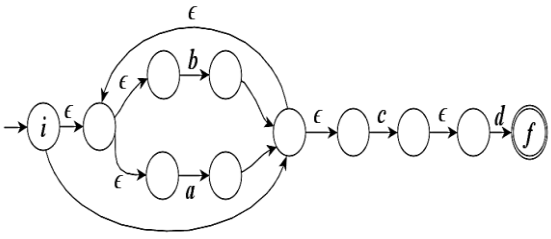

Figure 2: NFA for the regexp (*a*|*b*)*(*cd*) [15]**.**

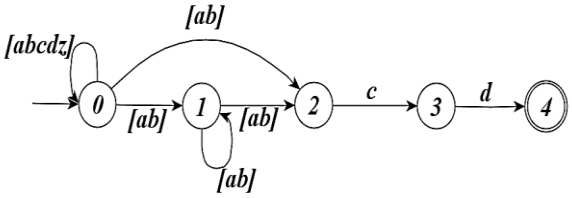

Figure 3: Minimized classified ECD-NFA with epsilon and self-transitions eliminated from 0 The letter *z* represent all those characters in the ASCII character set that are not *a,b,c,* or *d* (or simply put *[^abcd]*.

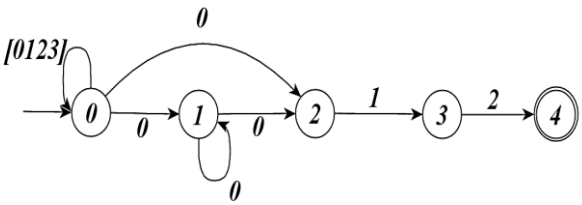

Figure 4: ECD-NFA now with assigned ECDs.

D. The Structure of the ECD-NFA

The block diagram as seen in 0 summarizes the flow of the entire ECD-NFA design. The design *phase 1* which is the software phase of the execution process is made up of blocks labelled as: Snort rulesets, PCRE regexps extractor, Parser, ECD-NFA constructor and optimizer, which is connected to units (a), (b), and (c). The last *phase 2* which is the hardware phase is made up of the block labelled as the Hardware ECD-NFA (FPGA) builder. The regexps are first extracted from the community Snort rulesets database [30] by the regexp extractor

and automatically fed into the parser for automation. During the process of constructing the ECD-NFA, an *n*-byte class table function is called to create the ECDs, where $n=2^k$, with $k=1$, and 2. The preliminary design generates results for a *1*-byte ECD matching design. Each of the matching units contain a single regexp. The structure as seen in 0 represents the overall ECD-NFA approach.

In order to optimize the ECD-NFA, the generated ECDs are used to drive the automata as seen in 0, which are now a compressed inputs. The unit labelled (b) in 0 performs the construction of the ECDs tables. The tables are later synthesized into a small piece of memory to perform table look-up operations against the compressed inputs of the unit labelled (a) seen in 0. The compressed ECDs are in BRAM format consisting of 256x8 bit inputs. These tables map each *1*-byte input against the corresponding ECD value.

Lastly, the BRAM module containing the ECDs together with the ECD-NFA are then translated from their Java-based code format to their equivalent VHDL codes by the VHDL function generator. The generated VHDL codes are saved and transferred into the hardware *phase 2* for synthesis. In *phase 2*, the process of VHDL code synthesis and translation into circuit descriptions or netlists takes place. Afterwards, the process of PAR converts the circuit design into actual bitstreams for loading into a target FPGA for configuration.

E. ECD-NFA Regular Expression Matching Circuit Block

The parser creates blocks of REMs used for matching regexps at once as earlier explained in Section II. Each REM block is made up of three sub-blocks. The first sub-block contains the ECDs in block labelled as (a) as seen in 0, while the second sub-block contains the decoding module used to decode the ECDs fetched from the BRAM. Once the ECDs are successfully identified, they are then released for onward passage in the form of equivalent *bit-vector* outputs. Each position in the bit-vector represents the required streamed ECDs. The bit-vector serves as inputs necessary to drive the third sub-blocks. The third sub-block consists of the ECD-NFAs generated as seen in 0.

Furthermore, *1*-byte of character input is fetched from the testbench module as seen in 0. The input is then used to look-up the equivalent ECDs from the BRAM block. Only the 7-bit value equivalent of the input is required to be decoded in the decoding module. The decoding module decodes the input using a simple decoding process and circuits. The output of the module is then released as a $\leq$127 bit vector, with each bit position representing each of the matched 128 ECDs fetched from the BRAM block.

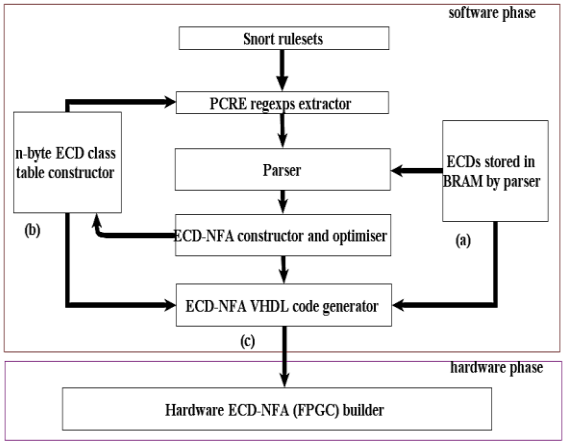

Figure 5: An inner look at the software and hardware toolchain flow for ECD-NFA design.

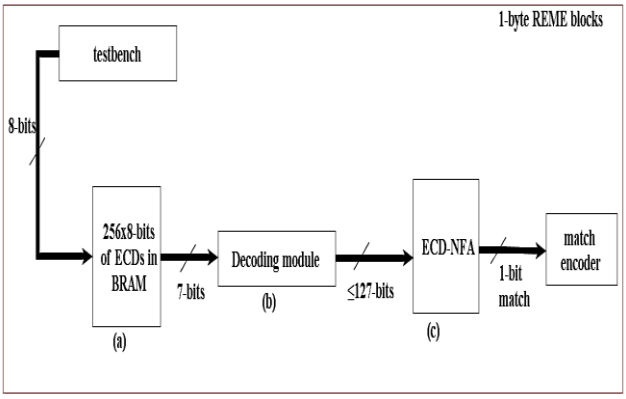

Figure 6: Block diagram of a *1*-byte input ECD-NFA REME matching block.

The bit-vector output from the decoding module is then passed unto the NFA module labelled as ECD-NFA to drive the automata for possible matching of the input strings. Each of the modular blocks in 0 is executed per clock cycle. The final output of the ECD-NFA module block is then encoded. To perform a matching across several REM blocks like the ones seen in Figure 6. The outputs stored as an encoded vectors of 1-bit outputs are logically ANDed together. Table 1 contains the result of our preliminary tests. The acronym *ECD-NFAnRE*, where $n = 1$ as seen in Table 1 simply refers to "an ECD-NFA compiled with *n* regexps", while the non-ECD-NFA is simply a naïve NFA design.

F. Evaluation

The ECD-NFA design is implemented using the Xilinx Vixtex-6 device synthesis tool. The ECD-NFA design requires only O(*n*) storage space for the ECDs and O(n^2m) time to process each of the ECDs extracted from each given regexp. It takes the design O(*nm*) time to search through a text of length *m*. A data bus width of 8-bits is used to compute the throughput measured in Gigabits per second (Gbps). But creating a design that generates high throughput while incurring minimal logic circuit cost has been a trade-off, and remains a challenge to REM-based designs. However, with clever algorithms, a balance can be achieved and the method for computing the throughput of matching is given as seen in Equation 1.

Equation 1: Computation for Throughput of Matching.

$$Throughput = (clock\ rate * data\ bus\ width)/1024 \quad [7] \qquad (1)$$

The column labelled *design* as seen in Table 1 represents the type of design approach. Also, the table column heading labelled *clock rate* (MHz) as also seen in Equation 1 represents the maximum clock frequency that each design attained. Lastly, the column labelled as *throughput* is the rate at which 1-byte characters are matched per clock cycle and is measured in Gigabits per second (Gbps). The throughput represents the speed at which some workload is accomplished. The data bus width as seen in Equation 1 is measured as 8-bit wide character. The product of the clock rate (MHz) and the data bus width (8-bits) divided by 1024 bits produces the throughput of matching in Gbps.

IV. Discussion

We shall quickly compare and contrast between the various designs as seen in Table 1, in order to have a diagrammatic view of how they compare against one another. There are two charts in this section, with each representing the relationship between the various designs and their respective reported clock rates and throughput as seen in Table 1. Figure 7 shows the various results for the clock rates reported against each design. Our ECD-NFA1RE design's clock rate is about 22% higher than the next highest reported clock rate by [28], and about 78% higher than the least reported clock rate by [24] as seen in Figure 7. The throughput of the ECD-NFA design is also about 22% better than the next highest throughput reported by [28] and as seen in Figure 8. This indicates that the ECD-NFA design shows some promise especially if it is properly developed. The results from Table 1 are used to generate the two charts in Section IV.

There is an explanation for the small throughput value 0.8 reported against their [24] design. The value of the throughput reported in [24] is the average throughput of each of the 16 separate 8-bit matching units that operate in unison. An 8-bit input is supplied to all the 16 matching units, which produce 1-bit output each or 16-bit output altogether. Preliminary results obtained for the non-ECD-NFA, ECD-NFA and the other related approaches needed to perform the comparative analysis in Section IV is reported in Table 1.

i. Table of Results

Table 1: Table of Design Results [7].

Design	Clock Rate (MHz)	Throughput (Gbps)
Non-ECD-NFA1RE	312.50	2.50
ECD-NFA1RE	460.00	3.68
Bispo et al. [28]	362.50	2.9
Clark and Schimmel [11]	250.00	2.00
Mitra, Najjar and Bhuyan [24]	100.78	0.8

ii. ***Chart for the Clock Rate***

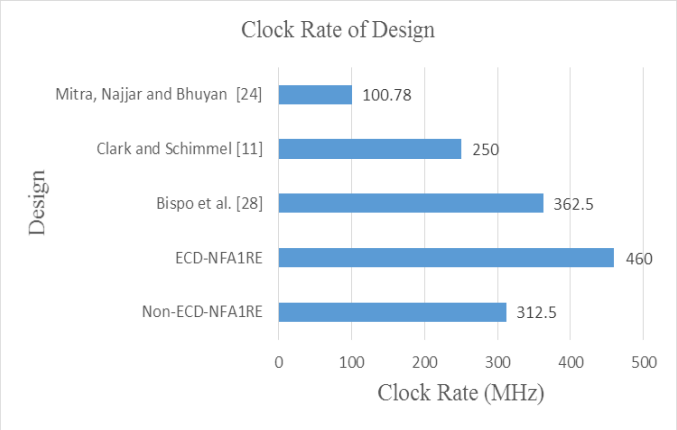

Figure 7: Designs against the clock rate (MHz).

iii. ***Chart for the Throughput***

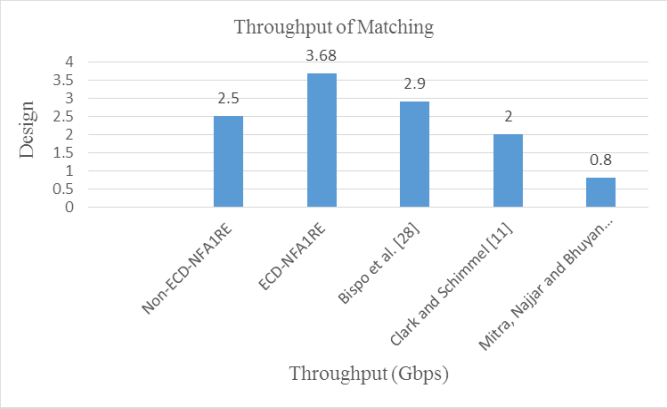

Figure 8: Designs against the throughput (Gbps).

V. Conclusion

Figure 7 and Figure 8 show some promise for the novel ECD-NFA approach. However, the biggest challenge with the current ECD-NFA matching circuit is that, the decoding module process described in Section III.E consumes too many BRAMs, shift registers and other logic circuits. The other challenge with the design is that it takes too long to synthesize and PAR. Notwithstanding, the ability to successfully compress inputs and synthesize their respective tables into a small piece of memory to perform table look-up operations is indeed a great breakthrough for the design. Furthermore, research is still ongoing to extend this work, by creating a multi-byte matching version of the design. The proposed newer version is to represent the more complex version of the ECD-NFA design. It is expected that the newer design will be capable of performing multi-character and multi-pattern matching in parallel.

As for the future work, there is a work in progress to be completed soon. The work seeks to address the complexity of the current ECD-NFA approach. The proposed approach will also create a design, capable carrying out parallel multiple-character and multiple-pattern matching. However, the issue of memory utilization will have to be addressed first to ensure efficiency in a separate work to be published. The general optimism is that the proposed multi-character and multi-pattern ECD-NFA design should be able to increase the current margin of the ECD-NFA design throughput by a scale of 4.

Furthermore, there is still the challenge of having to reduce the processing, synthesis and PAR time of the entire design. This will be necessary as the design becomes more complex. In order to achieve that, some optimisation technique need to be developed further by totally re-writing the current algorithm. However, the positive aspect is that the current design is able to generate a throughput of 3.68 at a clock rate of 460.00 MHz, which shows a great potential and promise. The design was written and implemented using the Java programming language for the software phase 1, and synthesized and PAR on a Xilinx FPGA Virtex-6 device synthesis tool for the hardware phase 2 of the design.

Acknowledgements

I acknowledge the contributions and support of Gerald Tripp towards the success of this ongoing research work. I also acknowledge the use of the Xilinx ISE Proprietary Project Navigator application version P.49d, volume 14.4 (nt64) Software. The software is bundled with the Xilinx Synthesis Tool (XST), which was used to implement this work.

References

[1]. J. Aycock, *Computer Viruses and Malware.* USA: Springer, 2006.
[2]. P. Piyachon and Y. Luo, "Compact state machines for high performance pattern matching, in *Proceedings of the 44th Annual Conference on Design Automation - DAC '07*, 2007, pp. 493-496.
[3]. M. Roesch, Snort - lightweight intrusion detection for networks, in *Proceedings of the 13th USENIX Conference on System Administration, LISA'99*, 1999, pp. 229-238.
[4]. (24 July 2013). *Bro.* Available: http://www.bro.org/download/index.html.
[5]. (24 July 2013). *IOS Intrusion Prevention System Deployment Guide.* [[Cisco IOS Intrusion Prevention System (IPS)] - Cisco Systems]. Available: http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6586/ps6634/prod_white_paper0900aecd8062acfb.html.
[6]. M. Becchi and P. Crowley, A hybrid finite automaton for practical deep packet inspection. in *Proceedings of the 2007 ACM CoNEXT Conference on - CoNEXT '07*, 2007b, pp. 1.
[7]. Y. E. Yang, W. Jiang and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA." in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems - ANCS '08*, 2008, pp. 30-39.
[8]. T. R. Bednar, P. H. Buffet, R. J. Darden, S. W. Gould and P. S. Zuchowski, "Issues and strategies for the physical design of system-on-a-chip ASICs " *IBM Journal of Research and Development*, vol. 46, pp. 661-674, 2002.
[9]. G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos and S. Ioannidis, "Gnort: High Performance Network Intrusion Detection Using Graphics Processors." *Recent Advances in Intrusion Detection*, vol. 5230, pp. 116-134, 2008.
[10]. G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos and S. Ioannidis, "Regular Expression Matching on Graphics Hardware for Intrusion Detection." *Recent Advances in Intrusion Detection*, vol. 5758, pp. 265-283, 2009.
[11]. C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks." in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004, pp. 249-257.
[12]. H. Wang, S. Pu, G. Knezek and J. Liu, "A modular NFA architecture for regular expression matching." in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays - FPGA '10*, 2010, pp. 209-218.
[13]. W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using FPGAs." in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures - SPAA '09*, 2009, pp. 188-196.
[14]. T. T. Hieu, T. N. Thinh, T. H. Vu and S. Tomiyama, "Optimization of regular expression processing circuits for NIDS on FPGA." in *2011 Second International Conference on Networking and Computing.* 2011, pp. 105-112.
[15]. R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs." in Proceeding FCCM '01 Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001, pp. 227-238.
[16]. J. E. Hopcroft, R. Motwani and J. D. Ullman , *Introduction to Automa Theory, Languages and Computation.* Boston, USA: Addison-Wesley, 2001.
[17]. R. Wain, I. Bush, M. Guest, M. Deegan, I. Kozin and C. Kitchen, "An overview of FPGA programming." vol. 1, pp. 4-5, 2006.
[18]. B. L. Hutchings, R. Franklin and D. Carver, "Assisting network intrusion detection with reconfigurable hardware." in *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* 2002, pp. 111-120.
[19]. P. Bellows and B. L. Hutchings, "JHDL-an HDL for reconfigurable systems." in *Proceeding FCCM '98 Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998, pp. 175-184.
[20]. (18 July 2013). *Snort IDS/Rules.* Available: http://www.snort.org/
[21]. L. D. Perry, *Programming by Example.* New York City: McGraw-Hill, 2002.
[22]. G. Tripp, "A Parallel "String Matching Engine" for use in High Speed Network Intrusion Detection Systems " *Journal in Computer Virology*, vol. 2, pp. 21-34, 2006.
[23]. C. Lin, Y. Tai and S. Chang, "Optimization of pattern matching algorithm for memory based architecture." in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems - ANCS '07*, 2007, pp. 11-16.
[24]. A. Mitra, W. Najjar and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS." in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems - ANCS '07*, 2007, pp. 127-136.
[25]. Y. E. Yang and V. K. Prasanna, "Software Toolchain for Large-Scale RE-NFA Construction on FPGA " *International Journal of Reconfigurable Computing*, vol. 2009, pp. 1-10, 2009.
[26]. B. C. Brodie, D. E. Taylor and R. K. Cytron, "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching." *ACM SIGARCH Computer Architecture News*, vol. 34, pp. 191-202, 2006.
[27]. P. Gupta and N. McKeown, "Packet classification on multiple fields." in Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication - SIGCOMM '99. 1999, pp. 147-160.
[28]. J. Bispo, I. Sourdis, J. M.P.Cardoso and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection." in *2006 IEEE International Conference on Field Programmable Technology*, 2006, pp. 119-126.
[29]. J. T. Arnold, "Lecture Notes for MATH 3034." pp. 1-13, 2007.
[30]. (18 July 2013). Snort IDS/Rules. Available: http://www.snort.org/.