NFA Based Regular Expression Matching on FPGA

Kamil SERT
Electrical and Electronics Engineering
Middle East Technical University, Ankara, Turkey
ksert@metu.edu.tr

Cüneyt F. BAZLAMACCI
Computer Engineering
Izmir Institute of Technology, Izmir, Turkey
cuneytbazlamacci@iyte.edu.tr

2021 International Conference on Computer, Information and Telecommunication Systems (CITS) | 978-1-6654-4913-7/21/$31.00 ©2021 IEEE | DOI: 10.1109/CITS52676.2021.9618426

***Abstract*— String matching is about finding all occurrences of a string within a given text. String matching algorithms have important roles in various real world areas such as web and security applications. In this work, we are interested in solving regular expression matching hence a more general form of string matching problem targeting especially the field of network intrusion detection systems (NIDS). In our work, we enhance a non-deterministic finite automata (NFA) based method on FPGA considerably. We propose to use a matching structure that processes two consecutive characters instead of one in order to yield better memory utilization and provide a novel mapping of this new architecture onto FPGA. The amount of digital circuitry needed to represent the NFA is reduced due to having less number of states and less number of LUTs in the devised 2-character regex matching process. An evaluation study is performed using the well-known Snort rule set and a sizable performance improvement is demonstrated.**

Keywords—regular expression matching, regex, string matching, NFA, network intrusion detection, network security

I. INTRODUCTION

A Network Intrusion Detection System (NIDS) detects malicious network packets by inspecting and comparing packet payloads with signatures in a given rule set. A signature-based NIDS inspects packet payload for any malicious pattern (signature) hence performing deep packet inspection (DPI) by generally employing a string matching technique. Regular expression matching is the general form of string matching and modern dictionaries such as SNORT includes regular expresiipons to match. Regular expression matching implementations on FPGAs are challenging since they have very limited fast memory. Current research, including the present paper, aims to increase match (search) speed and reduce memory to store signatures. FPGAs can make high-speed computations possible but their internal memories are currently limited to about tens of megabytes. We therefore aim to use this limited memory more effectively in order to reach higher throughput on the same size device.

Our objective is to obtain an NFA-based high-throughput memory-efficient regular expression matching engine (REME) targeting state-of-the-art FPGAs. We propose an NFA architecture and its associated circuits, which makes it possible to represent two consecutive characters as one state. We also present a possible mapping of the proposed architecture onto FPGA circuits. The new architecture requires less number of flip-flops to store the same number of regexes in comparison to previous studies. LUT usage in this new approach is also less than the previous approaches even in the worst case.

We evaluate the performance of our proposal by using regexes extracted from the Snort IDS ruleset and by comparing the observed test results with a previous study [7], which shows that the new approach performs considerably well.

The present work makes the following contributions: i) RE-NFA architecture proposed by Yang et al. [7] is modified and a different NFA, which includes additional transitions corresponding to 2-stride inputs, is constructed ii) six different circuit modules are created to easily translate the architecture onto FPGA circuits and algorithms to create HDL codes are developed for mapping the circuits onto FPGA [15] iii) centralized character classification, proposed originally in [8], is adopted in order to better utilize BRAM resources of the FPGA, which in turn helps in improving also the overall resource efficiency.

The rest of the paper is organized as follows. We discuss the background and previous works about regular expression matching (REM) in Section 2. We review the base architecture proposed by Yang et al. [7] in Section 3. Section 4 presents our novel 2C-NFA architecture and defines the corresponding circuit modules. Performance evaluation of the 2C-NFA architecture and the comparison of it with its original form [7] is provided in Section 5. Section 6 concludes the work.

II. BACKGROUND & LITERATURE REVIEW

Early deep packet inspection (DPI) techniques relied on exact string matching for attack detection [8]. Then other techniques followed that use regular expression matching (REM) [9] because regexes are more flexible in representing complex string patterns. Relevant REM works can be broadly categorized as NFA-based, DFA-based and hybrid designs. NFA-based implementations use less memory but their matching speeds are slow. Previous researchers have focused on utilizing parallelism to increase the matching performance of NFA based designs [10][11]. Hybrid techniques combine the benefits of DFA and NFA [12].

The study in [4] is the first practical NFA implementation on reconfigurable hardware. A character-comparator circuit, a FF and an AND gate are combined to implement a single character NFA match module and using this module union, concatenation and Kleene-Star operators are easily constructed. Other realizations exist [13][14].

III. COMPACT ARCHITECTURE FOR HIGH-THROUGHPUT REGULAR EXPRESSION MATCHING ON FPGA

In this part, Yang et al. [7] is reviewed since it forms the basis for the work in the present paper. Yang et al. [7] have already modified the original RE-NFA conversion approach used in [4] in order to get a modular structure to help in translating an NFA into FPGA circuits easily. They also suggested a simple way of using FPGA BRAM resources for the task in hand. In the present paper, we pick up the advantageous Yang et al. [7] approach and using further optimizations and enhancements aim to get a search engine with higher throughput and less hardware resources. Yang et al. [7] implemented their regular expression matching engine (REME) on FPGA in three steps. First, a regex is parsed into a tree to perform a post-order traversal of the regex; second, a modular NFA architecture is constructed using the modified

McNaughton-Yamada construction algorithm and third, the resulting NFA is mapped into HDL for FPGA implementation.

An example NFA constructed by McNaughton-Yamada algorithm is shown in *Figure 1*. Given a regex, original McNaughton-Yamada NFA construction algorithm generates an NFA, which has many intermediate nodes and unnecessary (ϵ)-transitions, as shown as white circles and dashed lines respectively, in the figure. Yang et al. [7] modified the McNaughton-Yamada algorithm to eliminate unnecessary nodes and ϵ-transitions to reduce the memory cost and obtain a highly modular architecture that is easy to map onto an FPGA. The NFA constructed by the modified McNaughton-Yamada algorithm is shown in *Figure 1*. In the resulting NFA, shaded elliptic areas (basic state blocks) are identical and these blocks can be implemented as a single module, e.g. an entity in VHDL, having one OR gate, one AND gate and one state register . In order to translate the NFA into a circuit, all we need is to connect basic state blocks in accordance with state transitions.

In [4], character match signals were obtained via 8-bit comparators which requires two 16x1 look-up tables (LUTs) and one AND gate for its implementation. If such comparators are used for all states, LUT usage increases dramatically. Therefore, storing character match signals on BRAMs instead of comparators (hence LUTs) is preferable, which also does not affect clock frequency badly. For example, to implement ***\d*** character class (i.e. any digit), we need ten 8-bit comparators. Such an implementation in practice requires huge number of LUT resources and the associated clock frequency may be very low.

In [7] character classes are implemented by storing character match signals directly in BRAM instead. To implement the regex in *Figure 1*, we need five different character match signals (character classes) for ***a,b,c,#*** and ***[ac]***. Hence in the associated BRAM, there are 5 columns, each corresponding to a unique regex and 256 rows for each ASCII characters, and a 5-bit output to be used in the circuit. For example; in order to implement character class ***[ac]***, ***1*** is stored at row 97 and 99 of the column corresponding to character class ***[ac]***. If BRAM input is ***a*** or ***c*** then BRAM outputs ***1*** from its ***[ac]*** column indicating the match results.

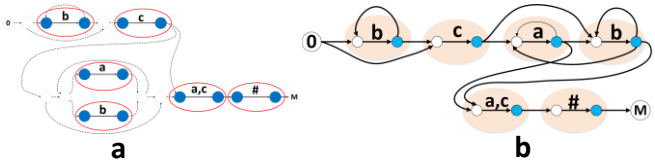

*Figure 1: NFA representation of **b*c(a+b)*[ac]#** for a) original b) modified McNaughton- Yamada construction*

In order to get higher throughput, Yang et al. finally proposed a multi-character input matching architecture. Multi-character inputs are also known as *strides* in the literature. In *Figure 2*, a 2-stride character matching circuit is shown. In comparison to single character matching, this approach requires nearly the same amount of LUTs but half the amount of state registers and one extra BRAM unit in order to obtain the character match signal for the second character. In the combined circuit, the first and second (blue and black lines in *Figure 2*, respectively) characters are processed by the lower and upper parts of the 2-stride matching circuit, respectively.

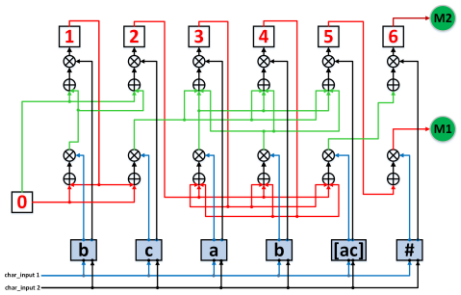

*Figure 2: Yang's 2-stride matching circuit for **b*c(a+b)*[ac]#***

IV. 2C-NFA Architecture for Fast Regular Expression Matching

In this section, we first discuss the main idea of our proposal and then present the design steps of the corresponding search mechanism to be constructed for finding all occurrences of strings that match the specified set of regular expressions and present six modules that are needed to implement the NFA-based circuit for this problem.

A. Main Idea

Our aim is to reduce the number of state registers and the number of AND/OR gates. Our idea is to represent compact state transitions for multiple consequtive characters in order to to reduce the number of state registers required in implementing the NFA. In this study, we propose a method to represent state transitions correspond to two consequtive characters, which is a case study for the above idea. Hence we represent the arrival of two consecutive characters in one state so that register usage will be reduced by at least 50% for a 2-stride matching circuit. This method combines registers 1-2, 3-4, and 5-6 into new ones in *Figure 2*. As a result we need 3 registers for the new case. This approach also prevents to use one extra OR gate for each of the regexes.

In the most general sense, Yang's architecture processes two characters at a time and for each character it needs one state register. On the other hand, our processes processes two characters at a time and for each two-characters it needs only one state register.

State transitions on finite automata corresponds actions triggered by a single character. For example; let a regex be ***kl(mn|op)qr,*** this expression can be matched by using the NFA in *Figure 3*. The automata goes from initial state ***S0*** to ***S1*** via single character ***k***, from **S1** to ***S2*** via ***l***, and so on. To store this regex on FPGA, we need eight state registers except the initial one. Our idea is to represent transitions as functions of two concatenated characters, that is transition from S_i to S_j can be triggered by a 2-stride. For example ***S0*** to ***S2*** transition occurs by receiving ***kl***. In this way, number of state registers required to store a regex on FPGA can be reduced by half. This approach will be referred as 2C-NFA in the rest of the paper. 2C-NFA representation for the example regex given above is redrawn in *Figure 3* by renaming inputs and states. To implement it, we now need only four state registers except the initial one.

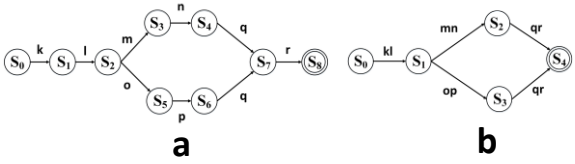

*Figure 3: a) Traditional and b) 2c-NFA for **'kl(mn|op)qr'***

B. Structural Construction of 2C-NFA

In this part, we develop the methodology to structurally construct the search mechanism that use 2C-NFA approach on FPGA. 2C-NFA architecture can be implemented in three steps as follows:

1. Given a regular expression, split it into sub-expressions using **(**, **)**, and | as delimiters.
2. Split each sub-expression into groups such that each group is composed of two non-star characters and including their following *'s, if exists.
3. Construct NFAs and their associated circuit corresponding to each group (hereafter modules) and combine them to form the final circuit.

1) Step 1

Any regex can be represented as composed of simpler sub-expressions. In this step, we identify **(**, **)**, and | as delimiters and label the remaining parts of the regex as sub-expressions. Let example regex ***RE*** be ***cde*f*(g*hij|kl*m*)nop***. RE can be re-written as the concatenation of the following sub-expressions having delimiters also in place.

$RE=RE_1(RE_2|RE_3)RE_4$ where:

RE_1: ***cde*f****, RE_2: ***g*hij***, RE_3: ***kl*m****, RE_4: ***nop***

2) Step 2

In this step, we find groups of character pairs in each sub-expression. For the given example, the groups are obtained as follows:

$RE_1: G_1G_2$ where G_1: ***cd*** G_2: ***e*f****

$RE_2: G_3G_4$ where G_3: ***g*h*** G_4: ***ij***

$RE_3: G_5G_6$ where G_5: ***kl**** G_6: ***m****

$RE_4: G_7G_8$ where G_7: ***no*** G_8: ***p***

Hence ***RE*** becomes $G_1G_2(G_3G_4|G_5G_6)G_7G_8$. Final group may be composed of a single character (i.e. G_6 and G_8).

3) Step 3

We observe that there can be at most 6 different types of groups for any regex if the above splitting mechanism (steps 1 and 2) is employed. General form of expressions corresponding to these possible groups are shown below:

- ab type-1 module 'T1' (G_1, G_4, G_7)
- a type-2 module 'T2' (G_8)
- a*b type-3 module 'T3' (G_3)
- a* type-4 module 'T4' (G_6)
- ab* type-5 module 'T5' (G_5)
- a*b* type-6 module 'T6' (G_2)

The implementation details of the above module library (T1-T6) is presented in Module Design section. Assuming we have the necessary modules, we choose the associated T_i for each group and combine them as illustrated in *Figure 4*.

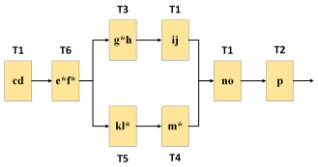

Figure 4: Combination of modules to implement a given regex

C. 2-Character Shifted 3-Character Window Search

In order to match a regex anywhere in the input stream with 2C-NFA approach, i.e. to eliminate false negative matches, it is necessary to present a new search mechanism suitable to the new architecture. In order to eliminate false negatives, we need three characters to check in parallel at every clock cycle while searching the input stream. Simply stated, we need to use 3 characters as input and then shift this 3 character window by 2 characters at every iteration since two input characters are to be consumed by the engine at each clock. The idea is illustrated in *Figure 5*.

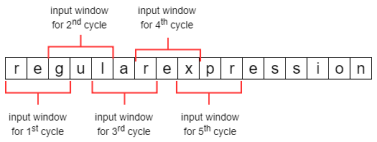

Figure 5: 2-character shifted 3-character window search

Match results will be obtained from BRAMs in our architecture, the same way as in Yang [7], but we need three replicated BRAMs to obtain the required character match signals. For every character match result, we need a corresponding 1-bit signal. BRAM outputs will be connected to the corresponding character match inputs of the state logic.

D. Module Design

1) T1 Module

T1 implements C_xC_{x+1} type patterns (i.e. C_x and C_{x+1} characters form a stride). With our search mechanism, we search for string C_xC_{x+1} in a 3-character window. $C_xC_{x+1}C_{any}$ and $C_{any}C_xC_{x+1}$ are two patterns to search by T1. NFA representation and the circuitry of T1 is shown in

Figure 6.

C_{any} means any character. When implementing $C_xC_{x+1}C_{any}$, if successor module is available, we need a connection from it's left character (C_{x+2}) match signal. In such a case, $C_xC_{x+1}C_{any}$ is implemented as $C_xC_{x+1}C_{x+2}$. Otherwise, $C_xC_{x+1}C_{any}$ is implemented as $C_xC_{x+1}C_{none}$. C_{none} means no connection is available. When implementing $C_{any}C_xC_{x+1}$, if predecessor module is available, we need a connection from it's right character (C_{x-1}) match signal. In such a case, $C_{any}C_xC_{x+1}$ is implemented as $C_{x-1}C_xC_{x+1}$. Otherwise, $C_{any}C_xC_{x+1}$ is implemented as $C_{none}C_xC_{x+1}$. The explanations for C_{any} and C_{non} is valid for all modules.

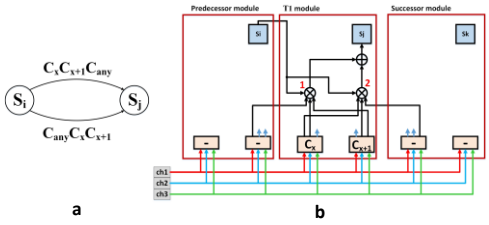

Figure 6: a) NFA representation of T1 b) T1 circuitry

2) T2 Module

T2 implements a single character C_x. T2 is similar to T1, but it can only be a final module (i.e. final state of a regex or sub-expression). $C_xC_{none}C_{none}$ and $C_{any}C_xC_{none}$ are the two patterns to search by T1 via proposed search mechanism. NFA representation and the circuitry of T2 is shown in *Figure 7*.

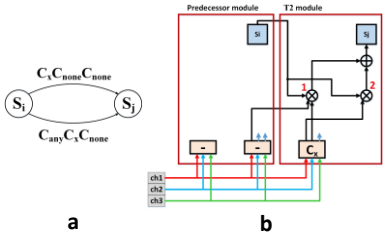

Figure 7: a) NFA representation of T2 b) T2 circuitry

3) T3 Module

T3 implements $C_x{*}C_{x+1}$ type regular expressions. Using $C_x{*}C_{x+1}$ we can derive C_xC_x and C_xC_{x+1} strides and C_{x+1} character. T3 has to search for them. Any string that finishes with C_xC_x stride should activate the S_i, therefore, $C_{any}C_xC_x$ pattern should be searched by T3. In order to find C_xC_{x+1} stride normally we search for $C_{any}C_xC_{x+1}$ and $C_xC_{x+1}C_{any}$ as in T1. But for T3 we have to search for $C_{none}C_xC_{x+1}$ instead of $C_{any}C_xC_{x+1}$. Due to *, we need such a regulation. In order to find C_{x+1} character $C_{any}C_{x+1}C_{any}$ and $C_{x+1}C_{none}C_{none}$ patterns should be searched. $C_{any}C_{x+1}C_{any}$ pattern also contains $C_xC_{x+1}C_{any}$, therefore we can combine them as $C_{none}C_{x+1}C_{any}$. Due to the nature of 3-character window search mechanism transition from predecessor of S_i (say S_h) to S_j is possible. Assuming that C_m and C_{m+1} are character classes of S_h, then there is a transition from S_h to S_j with $C_mC_{m+1}C_{x+1}$. NFA representation and the circuitry of T3 is shown in Figure 8.

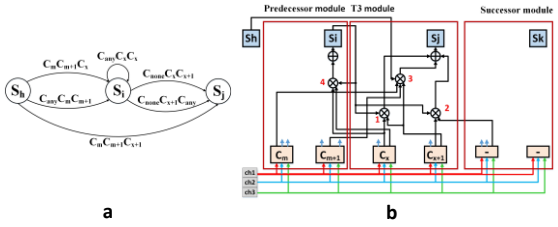

Figure 8: a) NFA representation of T3 b) T3 circuitry

4) T4 Module

T4 implements $C_x{*}$ type regular expression and is similar to T3. NFA representation and the circuitry of T4 is shown in Figure 9. We need to convert some transitions of T3 to implement T4. S_h to S_j via $C_mC_{m+1}C_{x+1}$ transition is converted into S_h to S_j via $C_mC_{m+1}C_{none}$. S_i to S_j via $C_{none}C_xC_{x+1}$ transition is converted into S_i to S_j via $C_{none}C_xC_{none}$. If S_i is an active state then S_j becomes an active state without any character match. This can be implemented by connecting output of S_i directly to OR gate of S_j.

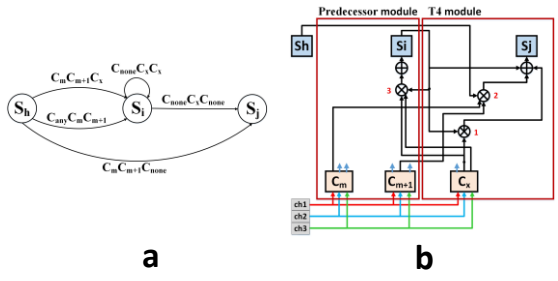

Figure 9: a) NFA representation of T4 b) T4 circuitry

5) T5 Module

T5 implements $C_xC_{x+1}{*}$ type regular expression. Using $C_xC_{x+1}{*}$, we can derive C_xC_{x+1} stride and C_x character. Any string that finishes with $C_{x+1}C_{x+1}$ stride should activate S_j, therefore, $C_{x+1}C_{x+1}C_{none}$ pattern should be searched by T5. To find C_xC_{x+1} stride, we have to search for $C_xC_{x+1}C_{none}$ instead of $C_xC_{x+1}C_{any}$. Due to *, we need such a regulation. Due to our new search mechanism transitions from predecessor of S_i to S_j and from S_i to successor of S_j are possible. NFA representation and the circuitry of T5 is shown in Figure 10.

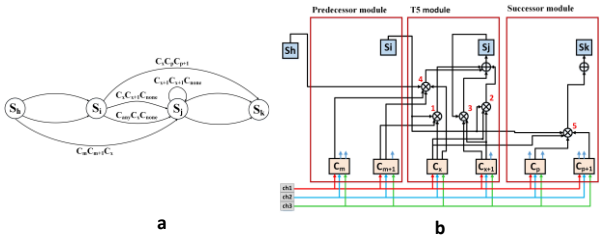

Figure 10: a) NFA representation of T5 b) T5 circuitry

6) T6 Module

T6 implements $C_x{*}C_{x+1}{*}$ type regular expression. T6 can be considered as a combination of T3 and T5 modules. So we can create T6 using transitions of T3 and T5. We can combine $C_mC_{m+1}C_x$ and $C_mC_{m+1}C_{x+1}$ as an $C_mC_{m+1}C_{none}$ pattern and $C_mC_{m+1}C_{p+1}$ and $C_xC_pC_{p+1}$ as an $C_{none}C_pC_{p+1}$ pattern. And also we can combine $C_{any}C_xC_{none}$ and $C_{none}C_{x+1}C_{any}$ as $C_{none}(C_x|C_{x+1})C_{none}$ in order to obtain a more compact circuit. NFA representation and the circuitry of T6 is shown in Figure 11.

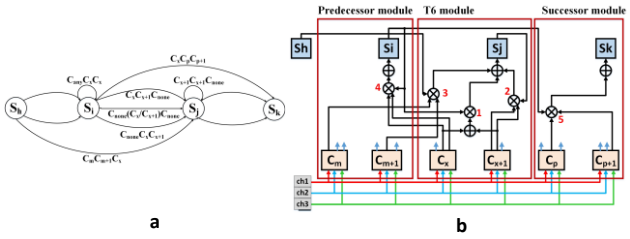

Figure 11: a) NFA representation of T6 b) T6 circuitry

V. Performance Evaluation

For evaluating the performance of the proposed architecture we extracted regexes from SNORT signatures. The aim of this work is to propose a general architecture to implement any regex hence all regexes in Snort are not used but a sample set is formed and used in our tests. While selecting regexes from Snort, we followed the following criteria: i) identical regular expressions stored in different rules are handled as a single one ii) regexes that are too short or containing repetition of one or more characters are avoided iii) regexes containing a large number of repetitions of a character or character groups are avoided.

We used 1052 regular expressions from 27 different categories of SNORT. State-of-the-art FPGAs has BRAM units that has 64-bit output. Since character classifications cannot fit in only one BRAM unit, we partition 1052 regexes into 11 different sets and we composed 3 different sets of regexes containing different number of regexes. 1052-reme set contains all regular expressions. *x*-reme set contains *x* regexes selected randomly. The aim is to see how scalable 2C-NFA is with respect to regex count.

We observe that 1052-reme runs with about 260 MHz clock and achieves a throughput of 4,16 Gbps. While achieving this throughput, we observe 11,3% LUT, 5,8% register, 13,3% slice and 12,5% BRAM unit usage. When this circuit is replicated 7 times, clock frequency of the circuit

decreases to 250 MHz and we obtain 28 Gbps throughput. We cannot replicate it one more time because FPGAs slice resources are exhausted and are not sufficient for further parallelization. 2C-NFA achieves maximum throughput of 51.5 Gbps when 569-reme set is replicated 14 times. Results of 2C-NFA and Yang et al. are given in TABLE *1* and TABLE *2*, respectively.

TABLE 1: IMPLEMENTATION RESULTS OF 2C-NFA

dataset	replication	LUTs	registers	slices	BRAM units	frequency (MHz)	throughput (Gbps)
1052 reme	x1	8916	9120	2618	33	260	4.2
	x7	62166	63841	17643	231	250	28.0
719 reme	x1	6603	6717	1939	27	260	4.2
	x10	69446	67289	18689	260	250	40.0
569 reme	x1	4899	5104	1430	18	260	4.2
	x14	68245	71443	18582	252	230	51.5

TABLE 2: IMPLEMENTATION RESULTS OF YANG ET AL.

dataset	replication	LUTs	registers	slices	BRAM units	frequency (MHz)	throughput (Gbps)
1052 reme	x1	9098	13018	3343	22	260	4.2
	x6	54595	78103	18948	132	250	24.0
719 reme	x1	7237	10069	2751	18	260	4.2
	x7	50756	70477	18060	126	250	28.0
569 reme	x1	5395	7483	1839	12	260	4.2
	x11	55122	82300	19052	132	243	42.8

We observed that place&route processes affects the reachable clock frequency. Synthesizers make optimizations to achieve higher clock frequencies when implementing the circuitry on a device. We also observed that; while implementing circuits, if BRAM units are not sufficient, the design tool utilizes from LUTRAMs, and as resource consumption increases on FPGA device, the clock frequency decreases. Minimum clock frequency, 230 MHz, is obtained when implementing 569-reme set with our 2CNFA architecture. Firstly, we compare memory costs for both architectures and see that LUT usage of 2C-NFA does not exceed Yang's and it nearly halves the number of registers required. We then analyzed the number of states that are used to represent datasets. Results are given in *TABLE 3*. 2C-NFA needs about 42% less number of states to represent datasets.

TABLE 3: NUMBER OF STATES TO REPRESENT DATASETS

	2C-NFA	Yang et al.
569-reme	6222	10975
719-reme	8141	14368
1052-reme	10530	18043

Because of the need for a 3-character window search mechanism, 2C-NFA needs 3-character match signals, therefore it needs 3 BRAM units. On the other hand, Yang's architecture needs 2 BRAM units. We create 11, 9 and 6 different character classes to implement 1052-reme, 719-reme and 569-reme, respectively. Hence to implement only one 1052-reme circuit, we need 33 and 22 BRAM units for 2C-NFA and Yang, respectively.

We may conclude that the limiting factor of 2C-NFA is BRAM usage while it is slice usage Yang's architecture. 2C-NFA achieves approximately 16%, 42% and 20% higher throughput than Yang's architecture for 1052-reme, 719-reme and 269-reme, respectively.

VI. CONCLUSION & FUTURE WORK

In this paper, we proposed a novel, modular, compact, NFA-based, memory-efficient, high-performance regular expression matching engine, which is suitable to be implemented on FPGAs. The memory-efficient nature of our approach helps us to implement more parallel circuits on a given FPGA device and therefore achieves higher throughput. 2C-NFA does not support runtime updates. While FPGA technology is continues to be improved, when more than 6-input LUTs are common in FPGAs, memory usage for 2C-NFA will be reduced even further in comparison to Yang's approach and hence larger throughputs will be possible in the future.

VII. REFERENCES

[1] Y. E. Yang and V. K. Prasanna, "Robust and scalable string pattern matching for deep packet inspection on multicore processors," IEEE Trans. on Parallel and Distributed Systems, vol. 24, no. 11, pp. 2283-2292, 2013.

[2] C. L. Lee and T. H. Yang, "A flexible pattern-matching algorithm for network intrusion detection systems using multi-core processors," Algorithms, vol. 10, pp.58-71, 2017.

[3] O. Erdem, "Tree-based string pattern matching on FPGAs," Computers and Electrical Engineering, vol. 49, pp. 117-133, 2016.

[4] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), pp. 227-238, 2001.

[5] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," IEEE Trans. on Parallel and Distributed Systems, vol. 25, no. 12, pp. 3088-3098, 2014.

[6] K. Peng, S. Tang, M. Chen and Q. Dong, "Chain-based DFA deflation for fast and scalable regular expression matching using TCAM," Seventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 24-35, 2011.

[7] Y. H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," Fourth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 30-39, 2008

[8] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," IEEE INFOCOM, pp. 2628-2639, 2004.

[9] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," Computer Communication Review, pp. 339-350, 2006.

[10] H. J. Kim and S.W. Lee, "A hardware-based string matching using state transition compression for deep packet inspection," ETRI Journal, pp. 154-157, 2013.

[11] J. Yang, L. Jiang, Q. Tang, Q. Dai, and J. Tan, "PiDFA: A practical multi-stride regular expression matching engine based on FPGA," IEEE International Conference on Communications, pp. 1-7, 2016.

[12] Y. Xu, J. Jiang, R. Wei, Y. Song, and H. J. Chao, "TFA: A tunable finite automaton for pattern matching in network intrusion detection systems," IEEE Journal on Selected Areas in Communications, pp. 1810-1821, 2014.

[13] D. Pao, N. L. Or, and R. C. Cheung, "A memory-based NFA regular expression match engine for signature-based intrusion detection," Computer Communications, pp. 1255-1267, 2013.

[14] T. T. Hieu, T. N. Thinh, and S. Tomiyama, "ENREM: An efficient NFA-based regular expression matching engine on reconfigurable hardware for NIDS," Journal of Systems Architecture, pp. 202-212, 2013.

[15] K. Sert, "NFA based regular expression matching on FPGA," MSc Thesis, Middle East Technical University, 2018.