HIGH-SPEED REGULAR EXPRESSION MATCHING ENGINE USING MULTI-CHARACTER NFA

Norio Yamagaki[†], *Reetinder Sidhu*[††], *Satoshi Kamiya*[†]

[†]System IP Core Research Laboratories
NEC Corporation
Kawasaki, Kanagawa, Japan
email: {n-yamagaki@cj, kamiya@ak}.jp.nec.com

[††]Applied Research Group
Satyam Computer Services Ltd.
Bangalore, India
email: Reetinder_Sidhu@satyam.com

ABSTRACT

An approach is presented for high throughput matching of regular expressions (regexes) by first converting them into corresponding Non-deterministic Finite Automata (NFAs) which are then configured onto a FPGA. The key novel feature is a technique that, for any given regex, constructs an NFA that processes multiple characters per clock cycle. An efficient algorithm is proposed that outputs an NFA which processes twice the number of characters as the input one. A technique is also proposed that implements the range match operation (e.g. [a-z]) efficiently. A program has been written that implements above ideas to convert regexes into NFAs specified in a structural Hardware Design Language (HDL), which are then mapped onto a FPGA. Performance is evaluated using real world regexes (Snort ruleset). The results demonstrate the practical utility of the approach. For example, for a set of 2,691 regexes, while the standard 1-character NFA obtains a throughput of 1.25 Gbps, our 4-character NFA achieves a throughput of 3.63 Gbps, while requiring only 20% more LUTs and 6% *less* flip-flops.

1. INTRODUCTION

Network bandwidths have been rising rapidly. At the same time, the frequency of network attacks and spam has been increasing. It is also becoming more important to provide varying Quality of Service (QoS) to different types of traffic. All above problems require packet processing, a key operation of which is searching packet contents for specified patterns which are typically specified as regular expressions (henceforth called "*regex*"). However, doing so at a throughput that matches network bandwidth, while crucial, has proven quite challenging.

Typically, on microprocessors, regex matching [1] is performed by first converting the given regex into a corresponding NFA or Deterministic Finite Automaton (DFA) which is then used to search input text characters. While a DFA can process each character in constant time (i.e. it requires $O(1)$ time), the number of DFA states, for an n character regex, can be $O(2^n)$ [2], which in some cases can significantly degrade performance. On FPGAs, on the other hand, regex matching can be performed using NFAs, again taking constant time per text character. And since NFA size is only $O(n)$ [2], the above problem is avoided. While NFAs can be used on microprocessors as well, doing so would require $O(n)$ time per input text character. On FPGAs, above time is reduced $O(1)$ by exploiting the available fine-grained parallelism which demonstrates a fundamental advantage FPGAs have over microprocessors for regex matching (for more details, please see Ref. [3]). While FPGAs have been used for simple string matching [4], the focus of this paper is on regex matching.

While the above NFA-based approach is quite efficient and throughput obtained is high, it is not high enough in the context of multi-gigabit wire speeds existing today and even faster speeds expected in the near future. So to improve throughput of above approach, some works [5][6] have been done on constructing NFAs that process multiple characters per clock cycle (henceforth called "*multi-character NFA*"). While improved throughput is shown for some examples, no procedure is provided for converting an arbitrary regex into a multi-character NFA.

This paper also proposes an approach for converting a regex into a multi-character NFA, but the approach is different from above previous works. Algorithms are provided to convert an arbitrary regex into an NFA capable of processing 2^k characters (for desired natural number k) per clock cycle. Perhaps as importantly, in comparison to previous works, the amount of additional logic required for multi-character NFA (relative to a 1-character NFA) is quite modest. In addition, a technique to generate efficient range match logic is also presented.

The above ideas are implemented in a program that outputs NFAs specified in a structural HDL which can be mapped onto an FPGA. The significant throughput improvement obtained using the proposed approach, using only a relatively small amount of additional logic is demonstrated by the results obtained for a few thousand real world regexes extracted from a Snort ruleset [7].

The rest of this paper is organized as follows: Section 2 presents the background. The proposed approach is described in Section 3, and the performance evaluation in Section 4. Finally, Section 5 concludes this work.

* This work was partly supported by Ministry of Internal Affairs and Communications (MIC) in Japan.

* The authors acknowledge the significant work assistance of Ashwini H. S. in implementation of the proposed approach.

978-1-4244-1961-6/08/$25.00 ©2008 IEEE.

2. BACKGROUND

First, we focus on the NFA-based regex matching logic on FPGAs. Next, we discuss related works using multi-character NFA so far.

2.1. NFA-based Regex Matching Logic

Sidhu, et al. propose a new regex matching logic design methodology [3]. They implement each state of NFA as a flip-flop, and its value shows whether the state is active or not. They also propose three basic NFA logic structures for the metacharacters of regexes '·', '|', and '*', and one logic structure for other characters. In the syntax tree, the non-leaf metacharacter nodes and the leaf character nodes are replaced by the corresponding logic structures yielding the required NFA logic. Figure 1 shows an example of NFA-based regex matching logic for "a(bc)*(d|e)", where the boxes *a* to *e* are character comparators. The simple and efficient NFA logic design enables high throughput. Since logic needs to be reconfigured when regexes are modified, the use of a reconfigurable device like a FPGA is required.

2.2. NFA-based Regex Matching Logic using Multi-Character NFA

In the above architecture, a single text character is processed in each clock cycle. On the other hand, to improve throughput, some works have been undertaken that process multiple characters per clock cycle.

Clark, et al. [5] propose a new architecture using multi-character NFAs, whose transition conditions (labels) consist of multiple characters. In this regard, however, to tackle wherever the character string of the label starts in the input string, the same number of NFAs as processing characters are required for a regex considering the offset. Therefore, although it can improve the throughput, its ability to do so is limited by the somewhat significant increase in logic size. Sutton [6] also proposes a new architecture using the similar manner to Clark, et al., the difference being parallel or sequential processing of multiple characters. This approach implements multiple character comparators between flip-flops. However, more the characters processed per clock cycle, the longer each path between flip-flops becomes. As a result, operating frequency would decrease, reducing the gain in throughput. Moreover, none of the above gives an effective procedure to construct a multi-character NFA for an arbitrary regex.

3. PROPOSED METHOD

We propose a novel logic design method using multi-character NFAs to realize high-speed regex matching. The method accepts several regexes as input, and outputs a description of the corresponding NFAs in HDL. That is, it is executed as preliminary step for the hardware configuration. This method consists of two key tasks;

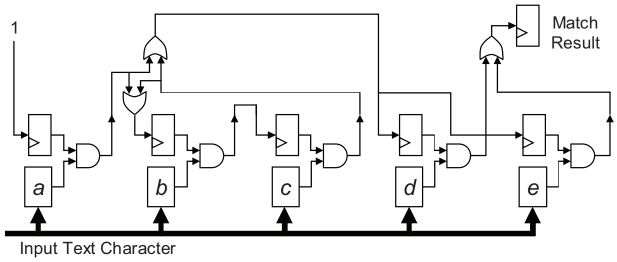

Fig. 1. NFA-based regex matching logic for "a(bc)*(d|e)".

NFA Construction: parsing of input regex into the syntax tree, conversion of the tree into an NFA structure, and modification of the NFA to support multiple characters per clock cycle.

HDL Generation: specification of the above multi-character NFA using structural HDL.

By performing the NFA construction task, our method can convert an input regex into the NFA which can process multiple (power of two) characters per clock cycle. Unlike the architecture presented in Ref. [5], our method constructs a single multi-character NFA which has almost the same number of states as an original 1-character NFA. If only the matching regex and not the exact match position is required, the constructed multi-character NFA has exactly the same number of states as the original NFA. In the HDL generation task, we also propose a technique to generate efficient range match logic. In this way, our method can configure the regex matching logic which can process multiple characters per clock cycle, and it can be expected to improve throughput. In the following, we explain the NFA construction task and the range match logic design.

3.1. NFA Construction

NFA construction task consists of two sub-tasks which are described in the following sections;

Phase 1: Conversion of a regex in post-order into its NFA graph that processes a single character every clock cycle.

Phase 2: Conversion of the above 1-character NFA graph into an NFA graph that can process 2^k characters (for desired natural number k) characters every clock cycle.

In phase 1, an input regex in post-order form is used to create the NFA graph corresponding to the regex. In order to perform this task for the various metacharacters, various graph operations have been developed. In phase 2, we propose a simple algorithm based on the transitive graph closure and execute a number of times depending on the number of processing characters per clock cycle. The resulting graph represents the NFA which can process the desired number of characters, 2^k, in parallel.

3.1.1. Phase 1: Regular Expression to 1-Character NFA

The input to the phase 1 of NFA construction task is a regex in post-order and the output is a 1-character NFA for the input regex. The regex in post-order can be easily obtained

```
for i = 1 to length(str)
  switch(str[i])
    case '|':  push(proc_or(pop(), pop()))
    case '·':  push(proc_and(pop(), pop()))
    case '-':  push(proc_range(pop(), pop()))
    case '?':  push(proc_ques(pop()))
    case '+':  push(proc_plus(pop()))
    case '*':  push(proc_star(pop()))
    default :  push(proc_char(str[i]))
```

Fig. 2. Pseudo code of phase 1 (NFA construction).

```
 1  add self edge labeled 'X' to initial node
 2  for each final node, add a new final node and
    connect former to latter by edge labeled 'X'
 3
 4  for n = each node in graph
 5    for i = each node having edge to node n
 6      for j = each node having edge from node n
 7        if(edge (n, j) ≠ self edge at line 1)
 8          add new edge (i, j)
 9          concat. labels of edges (i, n), (n, j)
10          add above label to new edge (i, j)
11
12  remove the original input graph edges, and the
    edges inserted at lines 1 and 2
```

Fig. 3. Pseudo code of phase 2 (NFA construction).

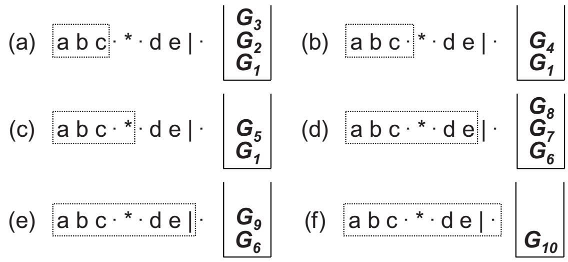

Fig. 4. An example of the stack in phase 1.

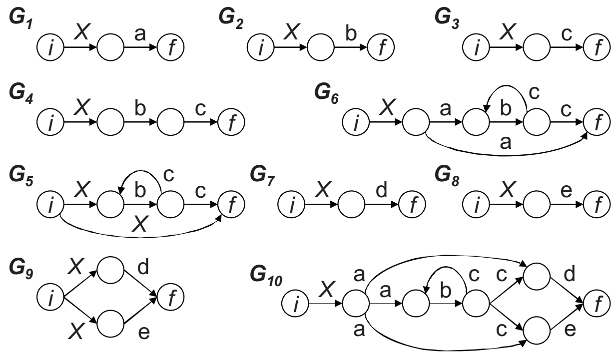

Fig.5. Examples of NFA graphs in phase 1.

by post-order traversal of its syntax tree. For example, the regex "a(bc)*(d|e)", in post-order form is "abc·*·de|·".

Figure 2 shows the algorithm. Although this algorithm is essentially the same as in Ref. [3], it has two additional functions for metacharacters '?' and '+'. In Figure 2, `str` is the post-order regex and a stack is used to store pieces of the NFA graph during processing. For example, the `proc_char` function constructs a simple NFA graph like the graph $\boldsymbol{G_1}$ in Figure 5. In the graph $\boldsymbol{G_1}$, there are three nodes and two edges, an edge labeled '*X*' (which denotes an arbitrary character) from the initial node to the intermediate one, and an edge labeled the character `str[i]` from the intermediate node to the final one. These functions accept one or two NFA graphs, and they all output an NFA graph, whose initial node has only output edges labeled '*X*' and final node has only input edges. The basic NFA graphs constructed by the functions for the metacharacters '·', '|', and '*' are based on Ref. [2].

Since `proc_star`, `proc_ques`, and `proc_plus` take $O(n)$ time and the other functions take $O(1)$ time, where n is the length of the regex, the algorithm requires $O(n^2)$ time.

3.1.2. Phase 2: 1-character to Multi-character NFA

In phase 2 of NFA construction task, construction of NFA graph for handling multiple characters per clock cycle is performed. The algorithm accepts a graph for an NFA that processes n characters per clock cycle and outputs a graph for an NFA that processes $2n$ characters per clock cycle. A 2^k-character NFA is thus obtained using k iterations of the algorithm. The algorithm is quite similar to the standard transitive graph closure algorithm. It is remarkable that such a simple algorithm enables a clean way of producing multi-character NFAs for arbitrary regexes. The algorithm is shown in Figure 3. It should be noted that the edges referred to on lines 5 and 6 are of the original input graph or those inserted in lines 1 and 2 only and not any of the new edges constructed by the algorithm. Since the algorithm doubles the number of final nodes, an NFA for 2^k characters will have 2^k final states, each final state corresponding to a character position, enabling multiple matches at different positions in the same clock cycle to be accurately reported.

If only information about whether the input string matches a regex or not is required (we call this "*non-match mode*"), following simplifications can be made; (1) replace line 2 by "`add self edge labeled 'X' to the final node`", and (2) modify line 7 to "`if( edge (i, n) ≠ self edge at line 2 and edge (n, j) ≠ self edge at line 1 )`". In this case, an output NFA has the same number of states as the input NFA. Each edge of a 2^k-character NFA is labeled with a string of length 2^k. For example, in Figure 6, transition from an active state along an edge labeled "bc" occurs only when the first and second input text characters in the current clock cycle are 'b' and 'c', respectively.

As each step takes $O(1)$ time, the algorithm requires $O(n^3)$ time, n being the number of states in the input NFA.

3.1.3. Example

We show an example of NFA construction for the regex "a(bc)*(d|e)". The post-order form "abc·*·de|·" is processed by phase 1 (Figure 2). Figures 4 and 5 show the stack and the NFA graphs $\boldsymbol{G_1}$ to $\boldsymbol{G_{10}}$, respectively. At the end of phase 1, the 1-character NFA graph $\boldsymbol{G_{10}}$ can be obtained.

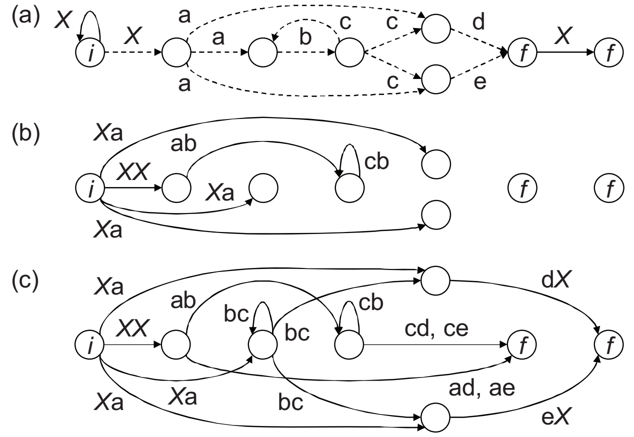

Fig. 6. Examples of 2-character NFA graphs.

Next, the above 1-character NFA is converted into the multi-character NFA by phase 2 (Figure 3). Figure 6 (a) shows the NFA graph obtained at line 2 in Figure 3, where dashed edges show original edges. Figure 6 (b) shows the half-constructed NFA graph after $n = 2$, where we omit the original edges. These tasks are performed for all of n, and then the 2-character NFA graph as shown in Figure 6 (c) can be obtained, where the right final state becomes active when a match occurs at the first position of the two input character positions, the left state becomes active when a match occurs at the second character position. If matches occur at both positions, both states become active. Similarly, we can obtain the 4-character NFA from above 2-character NFA. Thus, the 2^k-character NFA can be obtained by k iterations of phase 2.

3.2. Range Matching

The range matching regex matches a range of consecutive characters. For example, "[0-9]" matches any single numeric character. Efficient logic for range matching is obtained as follows. Consider an n-bit input I composed of bits x_{n-1} (MSB) to x_0 (LSB). Now consider the Boolean function $I \le C$ (C is an n-bit constant, $0 \le C \le 2^n-1$) which is 1 for all $I \le C$ and 0 otherwise. First, the Shannon decomposition of any Boolean function f_n of n inputs is;

$$f_n(x_{n-1},\cdots,x_0) = \bar{x}_{n-1}\cdot g_{n-1}(x_{n-2},\cdots,x_0) + x_{n-1}\cdot h_{n-1}(x_{n-2},\cdots,x_0) \quad (1)$$

If f_n represents $I \le C$, then exploiting the monotonic nature of f_n (in the truth table output column, all zeros are at the bottom), we can be derive;

$$f_n(x_{n-1},\cdots,x_0) = \begin{cases} \bar{x}_{n-1}\cdot g_{n-1}(x_{n-2},\cdots,x_0) & (c_{n-1}=0) \\ \bar{x}_{n-1} + h_{n-1}(x_{n-2},\cdots,x_0) & (c_{n-1}=1) \end{cases} \quad (2)$$

where c_{n-1} is the MSB of C. Similarly, for the $I \ge C$;

$$f_n(x_{n-1},\cdots,x_0) = \begin{cases} x_{n-1} + g_{n-1}(x_{n-2},\cdots,x_0) & (c_{n-1}=0) \\ x_{n-1}\cdot h_{n-1}(x_{n-2},\cdots,x_0) & (c_{n-1}=1) \end{cases} \quad (3)$$

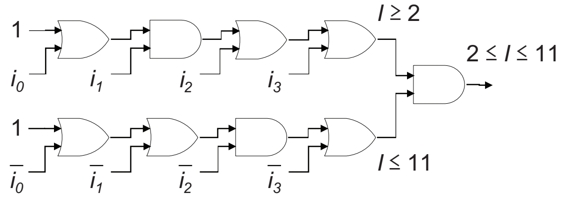

Fig. 7. Logic implementation of $2 \le I \le 11$.

Using the above equations recursively, one obtains efficient range matching logic. For example, for $n = 4$, the logic for $2 \le I \le 11$ is shown in Figure 7. For 8-bit characters, the above technique, which seems somewhat better than the one proposed in Ref. [8], enables range matching logic to be configured using only five 4-input LUTs (at most) for arbitrary ranges.

3.3. Prototype Implementation

We implement our ideas as a software tool, named *Regular Expression to Verilog NFA translator* (*REVN*). Its input, one or more regexes, are converted into 1-character NFAs, which are then converted into multi-character NFAs, which, specified in Verilog-HDL, is the output.

The input regexes to REVN are specified in the standard infix format, and conform to Perl-Compatible RegEx (PCRE) [9]. The metacharacters accepted are '*', '+', '?', '|', '(', ')', '[', '-', ']', '^', '$', '.', '\', '{', and '}'. REVN handles interval quantifiers, "{*n*}", "{*n*, }", and "{*n*, *m*}" in a straightforward manner, for example, "a{5}" is converted to "aaaaa". The characters accepted are any character with ASCII code from 0x20 (space) to 0x7e ('~'), the generic character types, '\d', '\D', '\s', '\S', '\w', and '\W', and the non-printing characters, '\a', '\e', '\f', '\n', '\r', '\t', and '\x' (character with hex code). The case insensitive match, and single line / multi lines match are also supported. Furthermore, REVN has an option which specifies *match mode* or *non-match mode*, as described in Section 3.1.2.

The HDL generation task essentially involves traversing the constructed NFA graph and for its nodes, edges and labels, specifying flip-flops, wires and combinational logic respectively, in structural Verilog-HDL. To configure efficient hardware logic in terms of logic size, the character comparators are shared among multiple transitions.

4. PERFORMANCE EVALUATION

In this section, the performance of regex matching logic constructed by our proposed method (using REVN) is evaluated by configuring it onto FPGA.

In this evaluation, to use meaningful regexes, we extract them from Snort 2.4 ruleset (unregistered user release) [7]. Concretely, we focus on "content", "nocase", "uricontent", "pcre", and "regex" options, and extract 2,691 regexes which do not include interval quantifiers and additional 357 regexes (3,048 regexes in all) which include them. We select 64, 128, 256, 512, 1,024, and 2,048 regexes

Table 1. Maximum operating frequency, *f*, and throughput, *T*, of 2,691 regexes (non-match mode).

#regexes	#chars	1-character NFA		2-character NFA		4-character NFA		8-character NFA	
		f [MHz]	*T* [Gbps]	*f* [MHz]	*T* [Gbps]	*f* [MHz]	*T* [Gbps]	*f* [MHz]	*T* [Gbps]
64	971	336.47	2.69	266.60	4.27 (158%)	183.35	5.87 (218%)	126.82	8.12 (302%)
128	1,955	271.96	2.18	246.37	3.94 (181%)	166.56	5.33 (245%)	103.92	6.65 (306%)
256	3,877	261.64	2.09	201.78	3.23 (154%)	160.51	5.14 (245%)	83.00	5.31 (254%)
512	7,803	224.16	1.79	184.09	2.95 (164%)	118.82	3.80 (212%)	84.42	5.40 (301%)
1,024	15,506	199.44	1.60	166.81	2.67 (167%)	124.86	4.00 (250%)	N/A	N/A
2,048	30,956	164.02	1.31	156.52	2.50 (191%)	110.06	3.52 (268%)	N/A	N/A
2,691	40,896	156.35	1.25	143.86	2.30 (184%)	113.38	3.63 (290%)	N/A	N/A

Table 2. Logic usage of 2,691 regexes (non-match mode).

#regexes	#chars	1-character NFA			2-character NFA			4-character NFA			8-character NFA		
		ALUT		Register	ALUT		Register	ALUT		Register	ALUT		Register
		Used	Util.	Used.	Used	Util.	Used.	Used	Util.	Used.	Used	Util.	Used.
64	971	938	1%	908	1,044	1%	916 (101%)	1,541	1%	932 (103%)	3,414	2%	964 (106%)
128	1,955	1,752	1%	1,721	1,856	1%	1,715 (100%)	2,524	2%	1,732 (101%)	5,459	4%	1,763 (102%)
256	3,877	3,231	2%	3,180	3,389	2%	3,190 (100%)	4,373	3%	3,204 (101%)	8,686	6%	3,237 (102%)
512	7,803	5,965	4%	5,881	6,341	4%	5,992 (102%)	8,046	6%	5,907 (100%)	19,595	14%	5,953 (101%)
1,024	15,506	11,421	8%	11,340	12,072	8%	11,640 (103%)	14,765	10%	11,327 (100%)	N/A	N/A	N/A
2,048	30,956	22,270	16%	22,161	22,697	16%	22,015 (99%)	26,943	19%	20,921 (94%)	N/A	N/A	N/A
2,691	40,896	28,401	20%	28,278	29,303	20%	28,379 (100%)	34,146	24%	26,636 (94%)	N/A	N/A	N/A

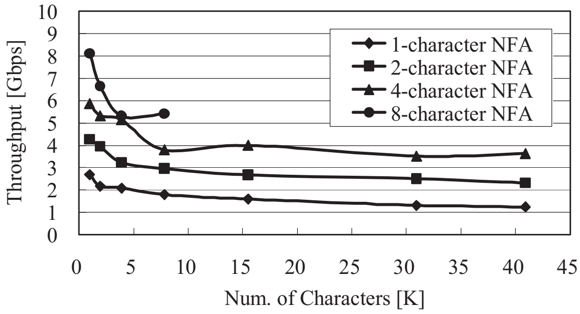

Fig. 8. Throughput of 2,691 regexes (non-match mode).

Table 3. Logic usage of 2,691 regexes (match mode).

#regexes	#chars	2-character NFA			4-character NFA		
		ALUT		Register	ALUT		Register
		Used	Util.	Used.	Used	Util.	Used.
64	971	1,060	1%	980 (64)	1,568	1%	1,124 (64)
128	1,955	1,939	1%	1,843 (128)	2,590	2%	2,116 (128)
256	3,877	3,572	2%	3,446 (256)	4,601	3%	3,972 (256)
512	7,803	6,767	5%	6,504 (512)	8,605	6%	7,443 (512)
1,024	15,506	12,978	9%	12,651 (1,011)	16,003	11%	14,410 (1,028)
2,048	30,956	24,825	17%	24,150 (2,135)	29,951	21%	27,055 (2,045)
2,691	40,896	31,716	22%	30,936 (2,557)	38,291	27%	34,781 (2,715)

randomly from the above two regex sets, and construct 1-, 2-, 4-, and 8-character NFAs for each group. We target Altera Stratix II (EP2S180) FPGA [10] and use Quartus II 7.2 SP1 [11] without any optimization options.

4.1. Experimental Results

Tables 1 and 2 show the maximum operating frequency, throughput, and the logic usage for the 2,691 regex set in non-match mode. Figure 8 shows the throughput for the same regexes. The throughput is calculated by multiplying the number of bits in characters processed every clock cycle by operating frequency. In Tables 1 and 2, #char (the number of characters) shows the total character count of the regexes except metacharacters, where the generic character types, the non-printing characters, and range match are counted as one character. Percentage within "(·)" shows an increase compared to 1-character NFA for each regex group, and "N/A" shows unavailable results because of very long compilation time in Quartus II. Table 1 and Figure 8 show that although multi-character NFA can improve throughput, the average increase is not proportional to the number of characters processed but approximately 170%, 250%, and 290% for 2-, 4-, and 8-character NFAs, respectively. Table 2 shows that the logic usage increases as the total character count is increased. It also increases as the number of characters processed is increased but the register usage is approximately constant. Although our method with non-match mode can construct multi-character NFA without change in the number of states, the multi-character NFA has slightly complex transition logic, which degrades operating frequency of the constructed multi-character NFA. In particular, the 8-character NFA logic usage seems disproportionately higher. This is due to the architecture of ALUTs, significantly more of which are required for transition logic for 8-character strings. Sharing in transition logic needs to be explored to reduce the logic requirements. The slight differences in flip-flop count are considered to be due to the optimization done by Quartus II.

Next, Table 3 shows the logic usage for the same set in match mode. In this mode, our method doubles the final states for each regex according to the number of characters processed. Ideally, the number of increased registers is calculated by $(m-1)\cdot N_r$, where m ($m \geq 2$) is the number of characters processed and N_r is the number of regexes. Each value within "(·)" shows the increase of registers per increased characters processed compared to the same multi-character NFA in non-match mode (Table 2). In Table 3, the extra registers is almost the same as the number of regexes. Due to them, the logic usage increases more than one in non-match mode. However, we confirm that the throughput shows similar results to non-match mode. That

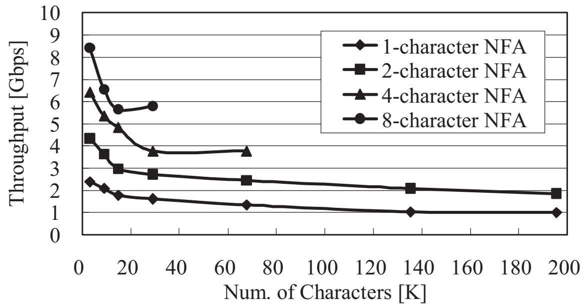

Fig. 9. Throughput of 3,048 regexes (non-match mode).

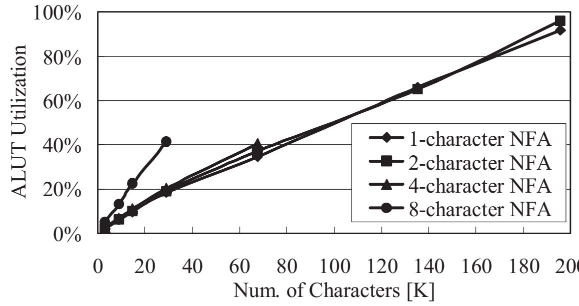

Fig. 10. Logic usage of 3,048 regexes (non-match mode).

is, the information of the matching position can be obtained without degradation of throughput by using our method.

Figures 9 and 10 show the throughput and logic usage for the 3,048 regex set in non-match mode. Due to REVN handling of interval quantifiers, this regex set includes more characters than the previous one. In fact, while the previous set includes up to 40,896 characters, this set includes up to 195,577 characters. In this case, although the throughput declines rapidly up to 20,000 characters, it does not do so from then on. The logic usage increases in proportion to the total character count. In the case of 195,577 characters, while the logic usage is more than 90% in 2- and 4-character NFAs, the throughput achieved is 1 and 2 Gbps respectively. Therefore, our method is expected to achieve high-speed regex matching.

Finally, we evaluate efficiency of each multi-character NFA by using *performance* [5] shown as Equation (4). This is a metric considering throughput and character density, and a logic with higher *performance* is more efficient logic.

$$\begin{aligned} Performance &= Throughput \times Density \\ &= Throughput \times Chars / LEs \end{aligned} \tag{4}$$

The number of Logic Elements (LEs) in Stratix II can be obtained as 1.25 times the number of ALUTs [10]. For 2,691 regexes in non-match mode, *performance* of 1-, 2-, and 4-character NFA are 1.44, 2.57, and 3.48 Gb/(s·LE), respectively. In addition, for 3,048 regexes in non-match mode, those of 1-, 2-character NFA are 1.20 and 2.10 Gb/(s·LE), respectively. In the other rule groups except 8-character NFA, similar trends are noted (8-character NFA shows similar values to 1-character NFA). Therefore, 4-character NFA is currently the most efficient.

5. CONCLUSION

In this paper, we proposed a novel regex matching logic design technique using multi-character NFAs. A simple algorithm for constructing such NFAs for arbitrary regexes was presented. Also, an efficient range match logic design technique is described.

Further, the proposed ideas were implemented in a software tool (REVN) and their utility was tested on a few thousand real world regexes. The results of performance evaluation show that our method can significantly improve throughput at only a relatively modest cost in terms of additional logic even without turning on optimizations while performing FPGA mapping. By turning on some of them, logic speed is likely to increase. Therefore, further throughput improvement can be expected.

Future directions to extend the work include reduction of logic size by sharing the states among multiple NFA, sharing transition logic, more detailed performance evaluations in various cases and on the actual system, and enhancement of PCRE support even further.

6. REFERENCES

[1] J. E. F. Friedl, "Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools," O'Reilly Media, Inc., Jan. 1997.

[2] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to Automata Theory, Languages and Computability, 2nd Edition," Addison-Wesley, Nov. 2000.

[3] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," in *Proc. 9th IEEE Symosium on Field-Programmable Custom Computing Machines* (*FCCM'01*), pp.227-238, April 2001.

[4] I. Sourdis and D. Pnevmatikatos, "Fast, Large-Scale String Match for a 10Gbps FPGA-based Network Intrusion Detection System," in *Proc. 13th International Conference on Field Programmable Logic and Applications* (*FPL'03*), pp.880-889, Sept. 2003.

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

[6] P. Sutton, "Partial Character Decoding for Improved Regular Expression Matching in FPGAs," in *Proc. 2004 IEEE International Conference on Field-Programmble Technology* (*ICFPT'04*), pp.25-32, Dec. 2004.

[7] http://www.snort.org/

[8] E. W. Spirznagel, "CMOS Implementations of a Range Check Circuit," Dept. of CSE, Washington Univ. Technical Report WUCSE-2004-39, July, 2004.

[9] http://www.pcre.org/

[10] Stratix II Device Handbook: Volume 1, available at http://www.altera.com/literature/hb/stx2/stx2_sii5v1.pdf

[11] http://www.altera.com/products/software/products/quartus2/qts-index.html