ECEB: Enhanced Constraint Repetition Block for Regular Expression Matching on FPGA

Le Hoang Long[1], **Tran Trung Hieu**[2], **Vu Tan Tai**[3], **Nguyen Hoa Hung**[4], **Tran Ngoc Thinh**[5], and **Dinh Duc Anh Vu**[6], Non-members

ABSTRACT

Recent Network Intrusion Detection Systems (NIDSs) utilize Perl Compatible Regular Expression to describe malicious patterns existing in the content payload of packets more and more efficiently. Several techniques are introduced to optimize the performance or complete the system for full support all of PCRE features in hardware platform, but some issues have just been solved partially.

Constraint Repetition is among important characteristics of PCRE but its effective hardware-based implementation solutions are still limited. This paper describes an Enhanced Constraint rEpetition Block (ECEB) for regular expression matching engine in FPGA.

To support more PCRE features, we improve our implementation to handle flag 'm' modifier. We also implement the block memory (BRAM) based character matching which saves a lot of LUTs and effectively improves overall system's throughput compared to related techniques. A software tool-chain for auto-generating PCRE matching system is also introduced. We do experiments on low-cost XC2VP50 Xilinx Virtex II Pro chip with the rule set of SNORT, an open source NIDS, to evaluate our implementation. The results verify that our architecture can achieve throughput up to 1Gbps and save up to 90% hardware resources compared with the conventional architecture.

Keywords: Constraint Repetition, FPGA, NIDS, PCRE

1. INTRODUCTION

The increasing attacks from the Internet like viruses, spam, malwares as well as other malicious activities give the rise to the protection methods which help protect users' private information, prevent from destruction as well. Due to various types of attack, the pattern of intruder can appear at any places of packet, not in certain specified location. Network intrusion detection system (NIDS) is one of the solutions for deeply inspecting all payloads of packet not only in the header like Firewall. Snort [1] is an open source NIDS which has thousands of rules defining attack signatures.

In NIDS, the escalation of time for processing and matching the patterns content of packets corresponds to the number of signatures that NIDS needs to process. Because of the requirements of quality of service, the speed of network reaches gigabits per second (Gbps) and maybe higher in the future. Software NIDS solutions just reach the bit rate of mega while every string of bytes of the traffic will be compared with a large number of rules, this string matching task is the main reason which makes software-based NIDS skip packet due to the limitation in speed. In addition, software solutions do not scale well with huge rule sets and their memory requirement may be significantly large. Due to the need to speed up the performance, hardware implementations appear as a necessary solution for high speed NIDS. The hardware solutions must have high degree of processing ability to mange complicated patterns; allow reconfiguration for changing and updating intrusion patterns. Hence, Field Programmable Gate Array (FPGA) offers a suitable hardware-based solution.

There are two types of content signature in NIDS: static pattern and Perl compatible regular expression (PCRE). While static pattern can only represent a single string, PCRE can specify a set of strings without listing all its elements. If a string is in the set described by one regular expression, we say that regular expression matches the string. Regular expression (regex) is written in a formal language with specified syntax. For clear understanding, we will list some basic syntaxes. The simplest regular expression is a single literal character. Apart from special metacharacters, characters match themselves. The alternation *(|)* is used to alternate two regular expressions to form new one. Besides, concatenation operator *(.)* concatenates two regular expressions. If $r1$ matches $s1$ and $r2$ matches $s2$, then $r1|r2$ matches s1 or s2. With same previous assumption, regular expression $r1r2$ matches $s1s2$. The quantifier *(*)* is repetition operator call Kleene-Star: r^* matches sequence of zero or more strings described by r.

Manuscript received on August 22, 2010 ; revised on , .

This paper is extended from the paper presented in ECTI-CON 2010.

[1,2,3,4,5,6] The authors are with Faculty of Computer Science and Engineering Hochiminh City University of Technology 268 Ly Thuong Kiet Street, Ward 14, District 10 Hochiminh City, Vietnam, E-mail: richardle1988@gmail.com,bamboolll@gmail.com, tantaitx@gmail.com,HungNguyen@cse.hcmut.edu.vn, tnthinh@cse.hcmut.edu.vn and anhvu@cse.hcmut.edu.vn

PCRE matching can be implemented using finite state machine (FSM). There are two types of FSM. Those are Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA). DFA allows only one active state per clock cycle whereas NFA offers more than one state that can prevent the state explosion and utilize the concurrent processing feature of FPGA. Moreover, when NFA approach is chosen, the smaller number of states is required compared to DFA solutions. Thus, NFA is more compact and area efficient than DFA. There are several researches implementing PCRE using NFA on FPGA platform. In 1999, Sidhu et al proposed basic NFA blocks on FPGA in [4]. Recently, researchers in South California University [7] proposed an new architecture for NFA blocks. With this approach, there is a minor difference compared with [4]. All flip-flops were placed after instead of before logic gates. Ioannis Sourdis and Stamatis Vassiliadis introduced three new basic building blocks to support constraint repetitions in [3].

Constraint Repetition is an important feature of PCRE and seen across practical rule sets. It allows a sub-regular expression repeating many times. Therefore, that may create a significant bottleneck for NIDS per-formance. The general constraint repetition quantifier specifies a minimum and maximum number of permitted matches by giving the two numbers in curly brackets (braces) separated by a comma. Table 1 shows the number of three types of constraint repetition in rule set of Snort 2.8.

Table 1: *Constraint Repetition in Snort 2.8. Three Types and Occurrence Times of Each Type Are In Rule Sets.*

Range	Atleast R{n,}	Exactly R{n}	Between R{n,m}
n<256	63	257	-
n<512	4	64	-
512≤n	41	26	-
Total	108	347	46

In order to implement constraint repetition on hardware, few approaches were proposed. The common conventional solution is using concatenation operator and union operator. In [3], Sourdis et al presented an implementation of *Exactly* block which takes advantage of the Xilinx SRL16 shift registers. But this method is only supports a single character or a character class. In [2], M. Faezipour et al presented SubRegex unit and a counter register for constraint repetitions. However, the implementation in [2] requires many resources for *Counter Reset Unit* which consists of multiple states identifying the mismatch. In addition, the implementation needs a supplemental Sub-Circuit for Atleast. Moreover, handling '^' operator and 'm' modifier flag is not mentioned in any previous designs. For that reasons, we proposes a new architecture based on novel architecture for regular expression matching in [7] to overcome these issues in [2] and [3]. Ours constraint repetition matching block can apply to any type of sub-regular expression with efficient hardware resources usage. In this paper, we also propose a technique that extends and improves the disadvantages of our previous work [8].

Our main contribution is proposing a new architecture enhanced with constraint repetition handling. We also handle '^' operator and 'm' flag for more full support PCRE features. In order to improve the efficient hardware resource usage of our design, we employ centralized pattern matching. We build a software tool-chain for auto-generating HDL code for PCRE matching engine.

The rest of this paper appears as follows. In section 2, we briefly cover some typical previous related researches on hardware-based regular expression matching also prior implementations for constraint repetition features. While section 3, we propose our improvement for repetition constraint support. The software tool-chain generating HDL code for PCRE matching engine with '^' operator and 'm' flags handling will be described in section 4. Within section 5, we will describe in details the improvement for ours system in [8]. Experimental results are provided in section 6. Finally, the conclusion is given out in section 7.

2. RELATED WORKS

From recent decades, there are several researches have been investigated the regular matching system in hardware. In 1982, Floyd and Ullman first introduced NFA model implemented on PLAs (Programmable Logic Arrays) [9]. In 1999, Sidhu and Prasanna proposed NFA basic blocks for Concatenation *(.)*, Kleene-Star *(*)*, Union *(|)* on FPGAs [4]. Clark - Schimmel et al used pre-decoding to share character comparators and thus reducing hardware resources along with throughput due to their capability of multiple characters processing per clock cycle [5]. Both methods become a model for most of later papers. Abhishek Mitra *et al*, presented NFA-based implementation by converting PCRE op-codes generated by Snort's PCRE compiler into VHDL, this method working resembles PCRE software engine [6]. Besides, Cheng-Hung Lin et al first showed the idea for sharing prefixes, suffixes and infixes. NFAs have most of common infixes and suffixes will be gathered into a group [10]. Recently, researchers in South California University [7] proposed an architecture for NFA blocks. With this approach, there is a minor difference comparing with [4]. All flip-flops placed after instead of before logic gates.

Constraint repetitions are special operators that indicate a sub-regular expression to be matched repeatedly for predefined number. The increasing of that feature in NIDS rule sets become main problem

for system implemented in hardware. One conventional solution to deal with them is unrolling which uses concatenation operator and union operator to replace constraint repetition. For instance, the Exactly block '$a\{100\}$' will be implemented by concatenating one hundred characters 'a' together. This approach is significantly wasteful. Recent solutions use NFA-based approach and build special block for constraint repetition. In [3], Sourdis et al presented an implementation of *Exactly* block which takes advantage of the Xilinx SRL16 shift registers to store multiple states and uses fewer FPGA resources. Since a SRL16 and a flip-flop can be mapped on a single logic cell, the implement for '$a\{100\}$' just takes less than 7 logic cells in Xilinx Virtex2 and Virtex4 devices. Due to cost area of O(N), this method is suitable for constraint repetition when N isn't too large. With larger value of N, this approach uses resources inefficiently. For instance, with $N = 2000$, this method needs 126 logic cells. Moreover, their approach just supports constraint repetitions of a single character or a character class which just take one clock circle to detect.

In [2] M. Faezipour and M. Nourani presented a novel NFA approach using a *Sub-Regex* unit for matching incoming data and a *Counter Reset* unit for counting the number of repetitions. With *Sub-Regex* unit, their approach can be applied to any sub-regex and was not suffering from overlapping conditions. However, their model requires redundant resources for Reset Unit and also need a additional circuit for *Atleast*.

To overcome the disadvantages of the architecture given by M. Faezipour and M. Nourani in [2], in ECEB, we offer a more efficient constraint repetition implementation especially in *Reset Unit*. Besides, we propose each model for each corresponding type to avoid the waste of hardware resource instead of using a common model for all types of *constraint repetitions*. This paper focuses our Constraint Repetition Block as well as our modifications to support common flags in PCRE. Besides, we used on-chip memory block (Block RAM) to match the incoming characters in place of utilizing the LUT, which offers more efficient hardware usage than ours previous work [8].

3. CONSTRAINT REPETITION ARCHITECTURE

This section will describe in detail all the units of constraint repetition block that we implemented. Figure 1 gives out the overview of the architecture.

SubRegex Unit matches an incoming string and generates an output signal to increase *CountComp Unit*. *Reset Unit* is responsible for resetting the counter inside *CountComp*. *CountComp Unit* counts the number of successive matching and compares the counter's value with pre-defined value to assert the output signal.

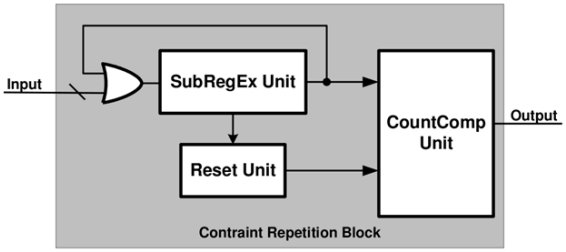

Fig.1: *Constraint Repetition Block Consists of Three Units: SubRegex, Reset, and CountComp Unit.*

3.1 SubRegex Unit And Reset Unit

SubRegex Unit is actually a PCRE matching subengine. Therefore, it can be applied to all patterns including a single character, a class of characters, or even strings containing constraint repetition and other meta-character. With this structure, all *State Blocks* (D flip-flops) are placed after logic gates. At every positive edge clock, if the output of previous flip-flop is active and incoming character is satisfied the condition of transition state, the output of next flip-flop will be active. When last flip-flop is active, string matches the circuit. Figure 2 depicts our *Constraint Repetition Block* for sub-regex '$ab + c$'. This sub-regex matches precisely n times a string which contains a character 'a' is followed by one or more characters b, and ends with single character c following immediately. With a set of incoming character '$abbbc$', D flip-flop will be active in order. First, D flip-flop matching character 'a'is active for one clock cycle. In next three clock cycles, D flip-flop 'b' will be active. Finally, D flip flop 'c' will mark a successive matching at next positive edge clock. The net connecting from the last AND gate to inc pin of *CountComp Unit* guarantees that the counter will be increased immediately before next character.

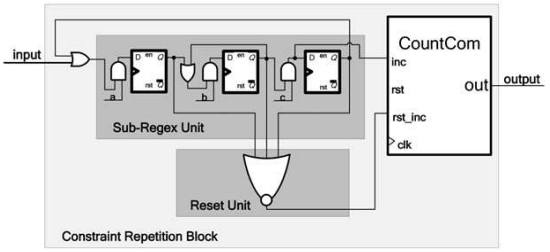

Fig.2: *Constraint Repetition Block for "$ab + c\{n\}$".*

Because this structure bases on NFA approach, multiple states (flip-flop) can be actived simultaneously. But when any mismatch occurs, all flip-flops are inactive (all outputs are in logic 0 level). With this feature, we propose a new model for Reset Unit that is more efficient than the architecture proposed in [2]. We just or all outputs with a K-input OR gate where K is the number of states in *SubRegex*

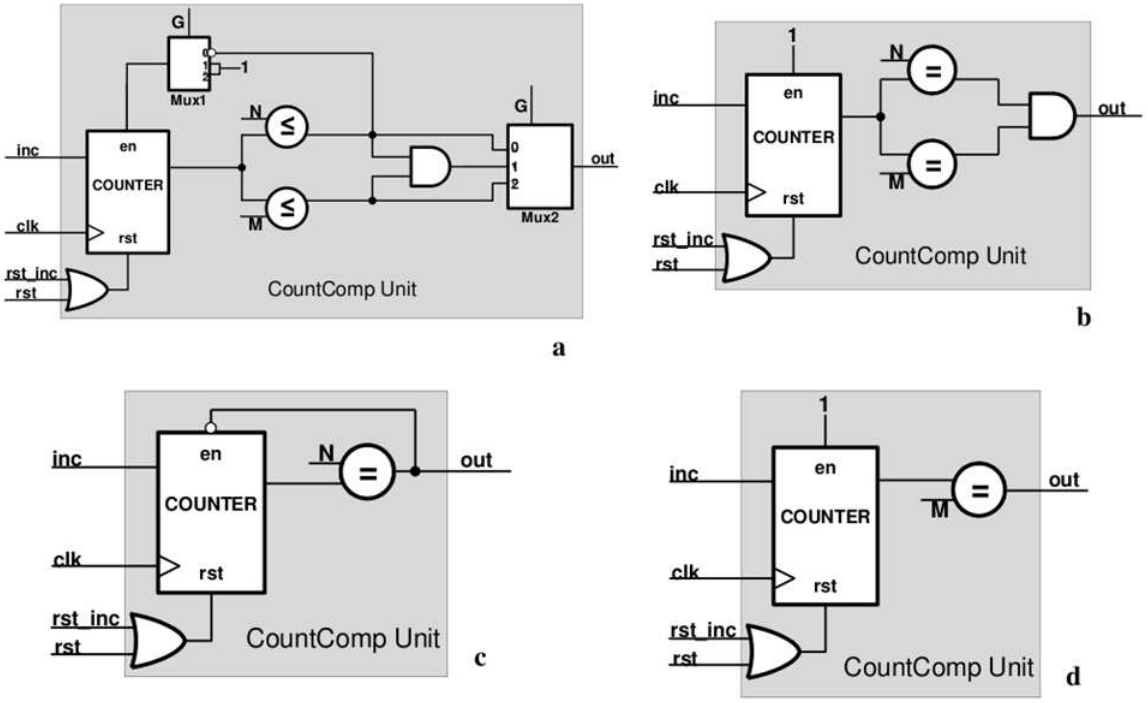

Fig.3: *CountComp Unit for each type of constraint repetition. (a) A standard CountComp Unit, (b) CountCompUnit for Exactly and Between, (c) CountCompUnit for Atleast, (d) CountCompUnit for Atmost*

Unit instead of generating the negation of intermediate states which can become complex. Whenever every mismatch takes place, the *rst_inc* input of *CountComp Unit* asserted then resets the counter to zero asynchronously.

3.2 CountComp Unit

CountComp Unit which is a form of counter has four inputs and only one output. It is described in Figure 3. At negative edge of clock, if *inc* is in high active, the value of counter is increased by one corresponding one time successive matching. When the maximum value of counter register is reached, the counter will be inactive and remain its value until the global reset signal (*rst*) or the local reset signal (*rst_inc*) becomes high. The *rst* signal comes from outside of Constraint Repetition is used for initialization when system is restarted or new data packet comes. *rst_inc* signal comes from Reset Unit whenever mismatch happens (as explained above).
As mentioned above, there were three types of Constraint Repetitions that can appear in matching pattern. To deal with this problem, we can construct only one counter block for all Constraint Repetition operators, then, give inputs to configure. Nevertheless, this causes a waste of hardware resources. Therefore, in our approach, we implement each type for corresponding Constraint Repetition. To do that, in module Verilog HDL, we declare parameters M, N and G to determine each specified type. Base on their values, synthesis tool will produce difference models which suitable for each type. Figure 3.a presents a standard structure for *CountComp* block. While Figure 3.b describes *CountComp* block handling *Exactly* and *Between*, Figure 3.c illustrates the model designed for *Atleast*. Figure 3.d shows the design for *Atmost* operator. In parsing PCRE process, the values of parameter M, N, G are assigned by our software tool-chain. Parameter M and N represent the range of Constraint Repetition. Parameter G is for determining type of Constraint Repetition. Table 2 lists entire capable values of M, N and G.

Table 2: *Values of M, N, G for Each Type of Constraint Repetition*

Type	Model	M	N	G
AtLeast	Regex{n,}	-	n	0
Exactly	Regex{n}	n	n	1
AtMost	Regex{,n}	n	-	2
Between	Regex{n,m}	m	n	1

In our design, the overlap problem will be handled when the NFA is generated from Parse Tree. Overlap problem is the condition when successive matching takes place in different positions of incoming string. These positions overlap the others. For instance, with an input string of '*aaaa*'and an sub-regex '*a*3', the successive matching can occur in two first positions of input string. The overlapping happens when constraint repetition operator occurs at the first position of regex. For instance, if the regex is '*a*3*b*' and the

Table 3: *Basic PCRE Syntaxes Are currently Supported by Our Tool.*

Operator	Name	Description
Regex1Regex2	Concatenation	Regex1 followed by Regex2
Regex*	Kleene - Star	Match Regex zero or more times
Regex1—Regex2	Union	Match Regex1 or Regex2
Regex?	Optional	Match Regex zero or one times
Regex+	Repetition	Match Regex one or more times
[abc]	Character Class	Match one character inside brackets
[âbc]	Negative Character Class	Match all character except characters inside brackets
Regex{n,m}	Constraint Repetition	Atleast: Regex{n,}, Exactly Regex{n}, Between {n,m}.
\xFF	Hex number	Match the ASCII character with the numerical value indicated by the hexadecimal number FF.
\d, \w, \s	PCRE shorthand character class	Match digits (0 to 9), word characters (digits and letters) and white space, respectively
\n, \r, \t	ASCII	Match LF, CR and tab, respectively
.	Dot	Match all characters except new line \n
[a-zA-Z]	Character Range	Match any characters in range
\?	Black slash	Escape meta-character
^	Caret	Match only at the beginning of a string
$	Dollar	Match only at the end of the string

incoming string is '*aaaaaab*' our counter register will be increased to six before meeting character b, which leads to mismatch case. To deal with this problem, we replace *Exactly* operator with *AtLeast* operator. After this step, the regex will become '*a*3,*b*' and it can exactly match incoming string of '*aaaaaab*'.

4. SOFTWARE TOOL-CHAIN

We implement a tool-chain to generate PCRE matching engine. Figure 4 shows an overview of our tool-chain. Generating PCRE matching engine contains three steps: (1) Parsing PCRE into a syntax tree structure. (2) Constructing a structural NFA with the McNaughton-Yamada algorithm. (3) Convert NFA into RTL code via Verilog HDL language.

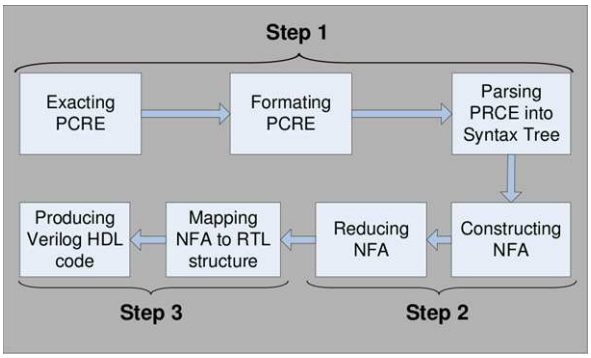

Fig.4: *Tool Chain to Generate Engine Contains Three Steps: PCRE Parsing, NFA constructing, RTL converting*

4.1 PCRE to Parse tree

The first step is representing each PCRE in corresponding parse tree. We utilize compiler technique to do that. There are two phases in this step. The first phase is pre-processing the PCRE. The tool-chain will parse PCRE from left to right and automatically insert a special character which is called char_and (its ASCII code is 176) to decompose PCRE into basic tokens. For instance, PCRE "$/a^*e(c \setminus d)^*[xyz] \setminus x3F/$", will become "$/a^{*\circ}e^{\circ}(c \setminus d)^{*\circ}[xyz]^{\circ} \setminus x3F/$". Table 3 lists entire basic PCRE operators supported to parse by our tool.

The second phase of this step is converting preprocessed PCRE to corresponding parse tree. Recursive constructing functions are invoked to parse each token. The resulted Parse Tree always consists of five types of internal nodes: op_and ($^{\circ}$), op_or (|), op_star (*), op_ques (?), op_plus (+) and leaf nodes. Each structural node of our tree has an extra component to identify its PCRE operator type (ID) which will be used in next steps. Figure 5 depicts the Parse Tree for PCRE "$/a^*e(c|d)^*[xyz] \setminus x3F/$".

4.2 PCRE Parse Tree to NFA

We apply McNaughton-Yamada algorithm [11] to construct NFA from parse tree generated in previous step as input. Base on McNaughton-Yamada algorithm rule and ID of NFA components, our tool traverses the parse tree recursively. At each node, the tool will call corresponding construction subroutine to build basic NFA block. Figure 6 shows the resulted NFA.

The NFA structure created from original McNaughton-Yamada algorithm has several useless states whose all transitions are unlabeled arrows (for instance, states

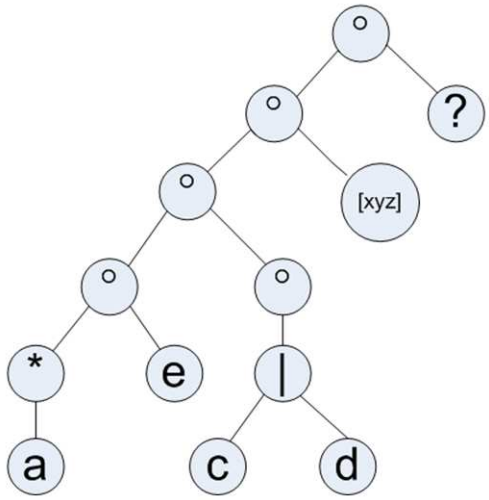

Fig.5: *Parse Tree for PCRE /a* e(c—d)* [xyz]\x3F/".*

1,4,7,8, 13 and 14). So these states can be eliminated to save resources. Figure 7 shows the NFA generated from a sub-function taking on deleting all redundant states.

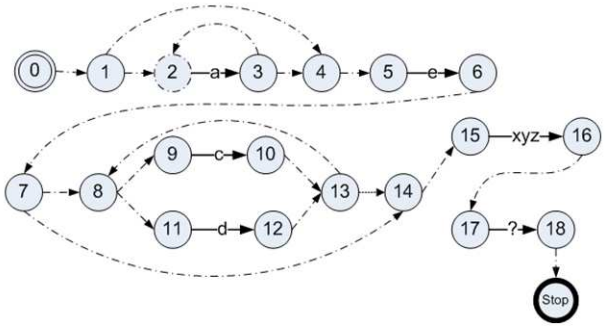

Fig.6: *NFA Structure Built from Original McNaughton-Yamada Algorithm for Parse Tree in Figure 5.*

4.3 NFA to HDL

In this step, we convert NFA to a suitable RTL structure proposed in [7]. As shown in Figure 7, all pairs of nodes inside ellipses have the same structure. They create a basic *State Block* for matching a single character. Each State Block is a Verilog module with only one difference parameter which is the number of input ports determined by number of immediately previous states having transition to the current state.

Most of *State Blocks* have the same structure except Start State Block (*State Block* number 0) and End State Block. *End State Blocks* do not accept any signal from character matching block. *Start State* (*State Block* number 0) is always active in PCREs containing no '^' meta-character, which ensures that

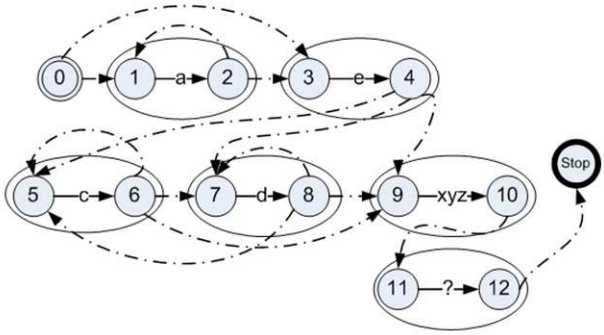

Fig.7: *The Reduced NFA Structure with The Number of States Is Reduced from 20 States to 13.*

our circuit be ready to match again if a mismatch case occurs at any time.

One n-input OR gate followed by an two-input AND gate just takes only four-bit LUT if n $\leq$ 3 or one slice if 4 $\leq$n$\leq$ 7 in FPGA devices have four-bit LUTs and two LUTs per slice. The circuit for matching PCRE: "$/a^*e(c|d)^*[xyz] \setminus x3F/$" is shown in Figure 9.

4.4 Flag *'m'* Handling

PCRE extends some flags such as '*s*', '*m*', '*i*'. These flags make PCRE be even more powerful to describe a matching pattern. Most of SNORT rules contain these flags, so dealing with them becomes more and more important. Either flag '*s*' or '*i*' can be easily handled whereas flag m requires harder handling. Therefore, in this sub-section, we will introduce our modification in *Start State* Block for full supporting matching PCREs which consist of operator '^' and flag '*m*'.

After parsing PCREs, our tool automatically constructs corresponding *Start State Blocks*. There are three difference types of *Start State Block* specified in figure 10. Unlike other *State Blocks*, the output of *Start State Block* is in high-level logic if and only if its input signal is in low-level logic. Figure 10a gives the model of Start Block for matching PCREs containing no operator '^'. PCREs require matching engine to be always willing to rematch a string after a mismatch happens at any character. This feature makes sure that our circuit can detect all substrings satisfying PCRE. The input of *Start State* in this type is always held in logic 0, which promises that the output is logic 1 at any given time. Figure 10b shows the model for *Start State Block* of PCREs consisting of operator '^'. Because their characteristic allows the circuit to start matching whenever the packet or string arrives, only when receiving a signal notifying that a new packet or string arriving, the output of our *Start State Block* is asserted. After that, its output is seized at low level until incoming next packet or next string actives it again.

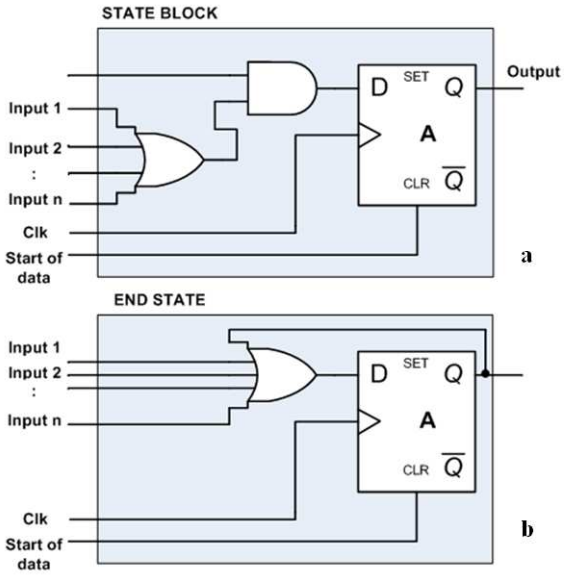

Fig.8: *(a) Normal State Block for Matching Single Character. (b) End State Doesn't Accept Any Signal*

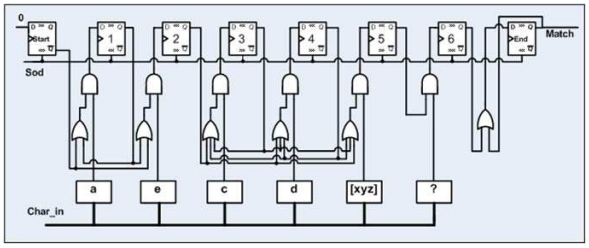

Fig.9: *Resulted Circuit Constructed by NFA from Figure 7 to HDL. Sod Input Denotes Start of Data Signal*

In addition, PCREs extend several new flags such as flags 's', 'm', 'i' that allow more flexible matching. If PCREs contains flag 'm', '^' then matching is included after new lines. For supporting this feature, we give third types of *Start State Block*. As shown in Figure 10c, the third model receives an inverted resulting signal generated by a *Char Block* being in charge of matching newline ($\backslash n$) character instead of holding a constant level logic (low in the first type and high in case of the second) at the input. When accepting a signal informing start of packet/string or an asserted signal generated from newline-matching *Char Block*, the output of Start State Block is high active. In other cases, it is low active.

5. BRAM CENTRALIZED CHARACTER MATCHING

In the basic state block module described in previous sub-section, the matching result of an incoming character is referred as one-bit signal to AND gate from one matching block. The state logic transition does not care about if result of matching a single char-

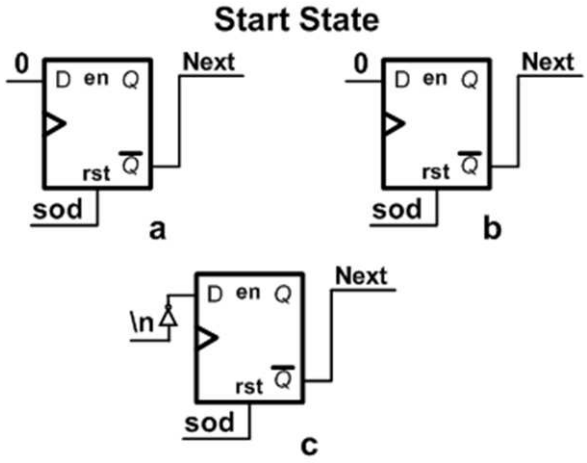

Fig.10: *Start State Block for PCRE (a) Has No Operator ^, (b) Has Operator ^, (c) Contains ^ and Flag m.*

acter or a class of character or any input symbol. The circuit does not distinguish matching between set of characters and single character or any general character range component of PCRE. All of them are alluded to a character classification block which takes an one-byte character input and generates a result bit vector to each corresponding state of PCRE matching engine.

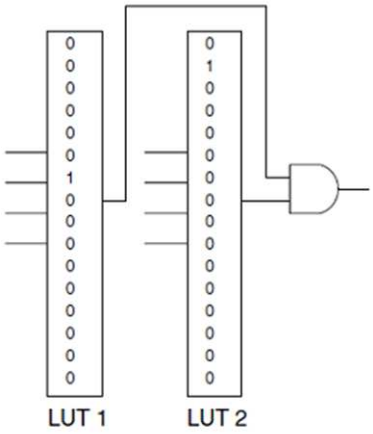

Fig.11: *LUT-based Character Matching Firstly Introduced [4] Requires Three Lookup Tables for One-Character Matching.*

In previous work [8], our character matching block was implemented by using LUT-based model which first introduced in [4]. This model is depicted in Figure 11, each LUT is equivalent to 4-input logic function, hence 8-bit character will cost 2 LUTs to present. There were two disadvantages when using this model to implement matching function. As shown in figure 9, the first is that we use one-character matching block for each state of system. In fact, if one engine consists of hundreds or even thousands of

character patterns, there will be the same number of identical full 8-bit comparisons performed. Besides, we do not implement these units cleverly, there may be in case that each engine has just its own block of comparisons and these are not shared with other engines. In that case, these redundancies waste logic and routing resources significantly. The second disadvantage is this method must broadcast the 8-bit incoming character to every comparison units of each engine. Obviously, thousands of comparison blocks require a lot routing effort to be done.

Paper [5] proposed a solution to overcome above problem. The input character will be decoded by shared 8-to-256 decoder into bit vectors whose bits are individually sent to corresponding unit's target character. However, this approach is not efficient with the character class comparison units and not fit with our architecture. We implement this block in BRAM on-chip memory of FPGA device with the general architecture depicts in figure 12. The incoming character matching is carried out by accessing the content entry of BRAM. This BRAM in this method is a block memory of 256 × n bits. The depth of BRAM is 256 and the width of an entry depends on the number of different characters in the set of given PCREs. Every clock cycle, one byte in payload of packet is introduced into system. System treats the value of this byte as an index for accessing correlative entry. Data read from an entry is a bit vector with every bit referred as resulting matching for each state. Subsequently, individual bits are automatically routed to relevant states of the PCRE matching circuit. As described, one n-state engine matching one PCRE, we only need a block memory of no more than 256 × n bits. Furthermore, if two states use the same character class for matching inputs, they can share the same output. In order to save hardware resources, BRAM is shared between many PCREs which have states accepting the same input character resulting signal.

In order to keep the fan-out of BRAM module low as well as maintain the clock rate and throughput of entire system, we utilize more than one BRAM. Each BRAM is just shared between a certain numbers of PCRE, which depends on our sharing policy. One important thing, the content of entire BRAM is pre-loaded when initializing and must not be changed during system's operation time.

Figure 13 describes the block diagram of matching system on FPGA. There are many PCRE engines which have capability of matching dedicated PCREs. The pre- interface and post-interface are responsible for communicating with external system. The pre-interface will accept data input, synchronize with control signal, and broadcast input character to every BRAM. The BRAM will be in charge of matching incoming characters and leading the output signal to correlative State Block in PCRE Matching Engines.

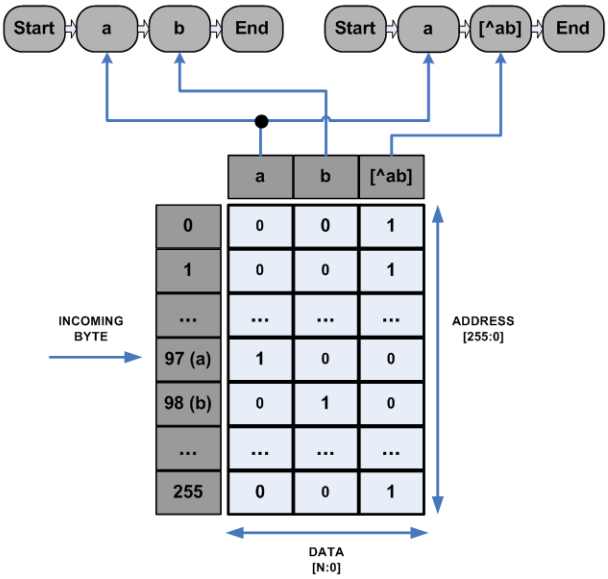

Fig.12: *BRAM Matching Model Use Incoming Byte As An Index To Generate Matching Bit Vector.*

The post-interface will accept the output bit vector of matching signal from PCRE Matching Engines, index it and send it to external system.

6. EXPERIMENTAL RESULTS

In this section, we present the experimental results on our PCRE matching system with the implementation of Constraint Repetition architecture along with BRAM Centralized Character. The result is also compared with some previous solutions. We take advantage of Xilinx ISE 10.1, a Xilinx Synthesis Technology (XST) tool for synthesis. In addition, the target FPGA device utilized for this experiment is Xilinx Virtex-II Pro XC2VP50 which consists of 23,616 slices and 4,176Kb Block Ram . In this implementation, we applied the PCRE patterns from Snort VRT rule set which releases on 22 July, 2010.

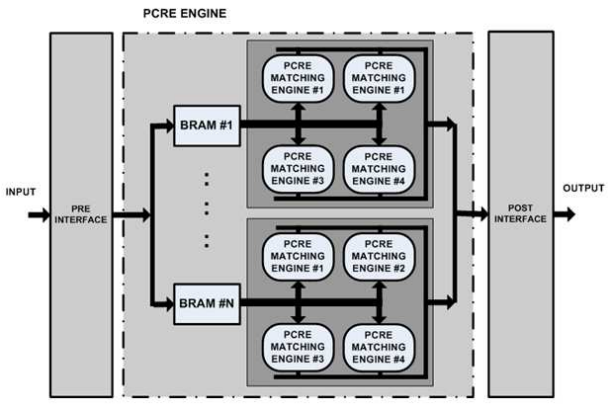

Fig.13: *Block Diagram of PCRE Matching System*

PCRE is introduced to Snort in order to increase

the accuracy of payload inspecting and the flexibility of pattern definition. However, the PCRE usage on software raises a performance problem, so every PCRE pattern in Snort rule is combined with Static pattern. For the purpose of PCRE matching system, we extract all PCRE pattern from Snort rule database and keep them in dedicated categories corresponding to original Snort rule sets. These PCRE patterns will be used by our software tool chain to automatically generate Verilog HDL code and send to XST for synthesis process.

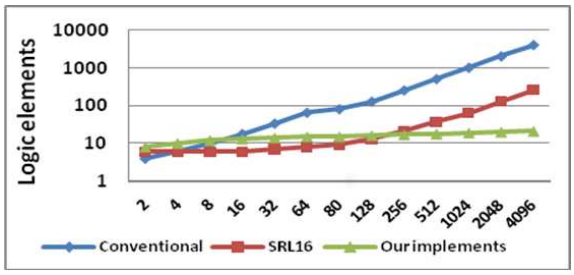

Fig.14: *Experiment Result of Three Constraint Repetition Implementation.*

In order to estimate the efficient of our Constraint Repetition block, we use the regular expression '$a\{N\}$'where N is come from 2 to 4096. Figure 14 gives the experimental result of three methods. The first one is Unrolling; this technique will concatenate the character 'a' in N times. The second one uses shift registers 16-bit as [2], and the last one is our approach. The diagram shows that when N is rather small, our approach costs more resources than others. However, when N is greater than 256, conventional approach grows rapidly, the SRL16 solution grows with slower speed. Our implementation shows outstanding capability of using hardware resources in that case.

Table 4: *Comparison between Conventional and Our Method to Handle Constraint Repetition Operator on Five Snort Rule Sets*

Rule set	Conventional		Our Implementation		Saved state (%)	Saved char (%)
	State	Char	State	State		
exploit	15999	15901	652	554	95.92	95.89
imap	6125	6043	590	508	90.36	91.59
smtp	5875	5801	508	434	91.35	92.51
voip	1662	1590	650	578	60.89	63.64
webmisc	6467	6425	512	470	92.08	92.68

Conventional implement uses unrolling in order to handle Constraint Repetition operator on FPGA. However, this method is definitely ineffective, especially with Snort Rule Set. Table 3 presents the comparison between our constrain repetition block with the conventional method. Two last columns clearly show that our approach uses hardware resources more effective than conventional method. In most of cases, ours saves up to 90% states and characters.
XC2VP50 consists of four megabits Block SelectRAMs, it is sufficient to implement our system with various numbers of BRAMs. By using BRAMs to store state transition character, we can save logic cells and reduce routing efforts. Hence, it will improve overall throughput of system. Table 4 presents our result on two character matching methods among different Snort rule sets: BRAM-based and LUT-based character matchings. The table clearly shows that BRAM method costs some memory bits, but saves large numbers of LUTs used. In addition, the supported clock cycle is always better than LUT method. Our system, therefore, can support Gigabit Ethernet.

Table 5: *Experiment Result of PCRE Matching System on LUT-based and BRAM-based Among Different Snort Rule Sets.*

Rule Set	LUT Base		BRAM Based			
	LUTs	ClK (Mhz)	LUTs	RAM (Kb)	ClK (Mhz)	Through-put (Ghz)
voip	872	125	736	19.712	161	1.288
webclient	1907	119	1711	29.44	128	1.024
webmise	858	142	737	25.088	145	1.160
oracle	572	118	448	20.224	145	1.160
smtp	841	138	664	24.32	166	1.328
backdoor	2955	134	2808	45052	157	1.256

7. CONCLUSION AND FUTURE WORK

In this paper, we presented the new Constraint Repetition architecture which saves up to 90% of States and Characters when applied to Snort Rule Set. The BRAM-based character matching is also implemented in order to replace original LUT-based approaches. This approach shows area hardware resources saving capability, also improves the system throughput significantly. We also introduce the new start block model which completely supports start operator '^' and 'm' flag. Moreover, we design software tool-chain to construct PCRE matching engines from list of given PCREs. Our tool can automatically analyze the list of given PCREs, reduce redundant characters and generate Verilog HDL module. Experimental results show that our system can sustain Gigabit network. Our future work will focus on solutions for back-references and techniques for better resource sharing.

References

[1] SNORT Network Intrusion Detection System, www.snort.org

[2] M. Faezipour, and M. Nourani, "Constraint Repetition Inspection for Regular Expression on FPGA," *16th IEEE Symposium on High Performance Interconnects*, pp. 111-118, 2008.

[3] I. Sourdis, S. Vassiliadis, J. Bispo, and J. M. P. Cardoso, "Regular Expression Matching in Re-

con?gurable Hardware," *Journal of VLSI Signal Processing*, pp. 1-23, 2007.

[4] R. Sidhu, and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 227-238, 2001.

[5] C. R. Clark, and D. E. Schimmel, "Scalable Pattern Matching for High Speed Networks," *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 249-257, 2004.

[6] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pp. 127-136, 2007.

[7] Yi-Hua E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 227-238, 2008.

[8] Le H. Long, Tran T. Hieu, Vu T. Tai, Nguyen H. Hung, Tran N. Thinh, and Dinh D. A. Vu, "Enhanced FPGA-based architecture for regular expression matching in NIDS," *the 7th IEEE International Conference Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-Con 2010)*, pp. 666 - 670, May 19-21, 2010.

[9] Robert W. Floyd and Je_rey D. Ullman, "The Compilation of Regular Expressions into Integrated Circuits," *J. ACM (New York, NY, USA)*, vol. 29, ACM, 1982, pp. 603-62.

[10] C.-H. Lin, C.-T. Huang, C.-P. Jiang and S.-C. Chang, "Optimization of Pattern Matching Circuits for Regular Expression on FPGA," *in IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 12, pp. 1303-1310, Dec. 2007.

[11] R.McNaughton, H. Yamada, "Regular Expressions and State Graphs for Automata," *in IEEE Transactions on Electronic Computers*, EC-9(1), pp. 39-47, 1960.

Le Hoang Long is currently studying toward his B.Eng degree in Computer Engineering from Ho Chi Minh City University of Technology (HCMUT) Vietnam, Faculty of Computer Science and Engineering. His research interests include Network Security on FPGA devices and Embedded Application.

Tran Trung Hieu is studying at Faculty of Computer Science and Engineering (CSE), The University of Technology, Ho-Chi-Minh city (HCMUT). He currently works toward his B.Eng degree in Computer Engineering. His research interests include Network Intrusion Detect System on FPGA and Embedded applications.

Vu Tan Tai was born in Nghe An, Vietnam, in 1988. He is studying toward his B.Eng degree in Computer Engineering from Ho Chi Minh City University of Technology, Vietnam, Faculty of Computer Science and Engi-neering. His research interest includes Regular Expression processing in hardware-based Network Instruction Detection System.

Nguyen Hoa Hung was born in Lamdong, Vietnam, in 1986. He received B.E. from Hochiminh City University of Tech-nology, Vietnam, in 2009. Since 2009, he was with the Faculty of Computer Science and Engineering, Hochiminh City Uni-versity of Technology, Vietnam, as a lecturer. His research interests include developing embedded system base on T-En-gine/T-Kernel Platform and Regular Expression processing in hardware-based Network Instruction Detection System.

Tran Ngoc Thinh received the ME and PhD from the King Mongkut's Institute of Technology Ladkrabang, Thailand in 2006 and 2009, respectively. He is now a lecturer at the Department of Computer Engineering, Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, Vietnam. His research interests include network security and bioinformatics on reconfigurable devices. He is a member of IEEE.

Dinh Duc Anh Vu is a senior lecturer in the Faculty of Computer Science and Engineering (CSE) at the University of Technology, Ho-Chi-Minh city (HCMUT) where he leads the Embedded Systems Group. His research interests include design automation of embedded systems, hardware/software verification, VLSI CAD, and reconfigurable architectures. Dr. Anh-Vu Dinh-Duc received the Master and Ph.D. degrees in Microelectronics from the Institut National Polytechnique de Grenoble (INPG), France, in 1998 and in 2003, respectively. He has been with the HCMC University of Technology (HCMUT) since 1995. Dr. Anh-Vu Dinh-Duc currently serves as a program/organizing committee member of several ACM and IEEE conferences. He is a member of the IEEE.