IPSJ Transactions on System LSI Design Methodology Vol.7 110–118 (Aug. 2014)

[DOI: 10.2197/ipsjtsldm.7.110]

Regular Paper

An Area Efficient Regular Expression Matching Engine Using Partial Reconfiguration for Quick Pattern Updating

Yoichi Wakaba[1,a)] Shin'ichi Wakabayashi[1,b)] Shinobu Nagayama[1,c)] Masato Inagi[1,d)]

Received: December 6, 2013, Revised: March 14, 2014,
Accepted: April 28, 2014, Released: August 4, 2014

Abstract: This paper proposes a method using partial reconfiguration to realize a compact regular expression matching engine, which can update a pattern quickly. In the proposed method, a set of partial circuits, each of which handles a different class of regular expressions, are provided in advance. When a regular expression pattern is given, a compact matching engine dedicated to the pattern is implemented on FPGA by combining the partial circuits according to the given pattern using partial reconfiguration. The method can update a pattern quickly, since it does not need re-design of a circuit. Experimental results show that the proposed method reduces 60% circuit size compared with the previous method without increasing the pattern updating time significantly.

Keywords: regular expression matching, partial reconfiguration, FPGA, systolic algorithm

1. Introduction

Regular expression matching (REM) is to find all substrings in a text, which match with a pattern described as a regular expression (RE) [1]. In database retrieval and network intrusion detection systems (NIDSs), fast REM for a large text is required. For example, NIDSs have to detect malicious packets (e.g., computer viruses) on a Gigabit speed network with thousands of virus patterns. Therefore, hardware implementation of REM has been widely studied. The existing methods of hardware implementation of REM can be classified into two approaches.

One is a pattern dependent approach [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. The pattern dependent REM engines have a dedicated circuit structure for a given pattern. The pattern dependent REM engines can realize fast REM with a compact circuit size. However, they cannot update a pattern immediately, since they need re-synthesis and reconfiguration of a circuit whenever a pattern is updated. This can be a significant disadvantage for NIDSs, in which patterns are frequently updated, since NIDSs cannot detect new computer virus while pattern updating.

The other is a pattern independent approach [13], [14], [15], [16], [17], [18]. The method can update a pattern immediately, since it does not need re-synthesis and reconfiguration of a circuit for a updated pattern. However, the size of a REM engine implemented to handle any RE patterns becomes large [18]. To reduce the circuit size, in Refs. [13], [14], [15], [16], [17], only RE operators often used in patterns are implemented, so that REM engines can handle a restricted class of REs. However, in recent years, since more complicated virus patterns have become popular, it has become difficult to describe virus patterns using restricted classes of REs.

In this paper, to achieve both the compact circuit size and quick pattern updating, we propose a method using partial reconfiguration as the third approach of REM hardware implementation. In the proposed method, a set of partial circuits, each of which handles a different class of REs, are provided in advance. Given an RE pattern, a compact matching engine dedicated to the pattern is automatically produced by combining the partial circuits according to the given pattern. Then, the produced engine is implemented on an FPGA using partial reconfiguration. The proposed method can produce more compact REM engines than pattern independent engines designed by the existing methods to handle any RE patterns. In addition, the proposed method can update a pattern more quickly than existing pattern dependent methods, since it does not need re-design of a circuit resulting in a long updating time. Experimental results show that the proposed method decreases 60% circuit size without increasing pattern updating time compared to an existing method [18].

The rest of this paper is organized as follows. In Section II., we explain REs, partial reconfiguration, related works and the systolic algorithm based REM engines. In Section III., the proposed method is shown. Section IV. shows the evaluation results. Finally, the conclusions are presented in Section V.

2. Preliminaries

2.1 Regular Expressions (REs)

RE is a method to represent a set of strings as a single string. A set of strings represented by an RE is called the regular language. We define REs as follows.

[Definition 1]

Let $\Sigma=\{a_1, a_2, \cdots, a_s\}$ be a finite set of symbols (alphabet). Let

[1] Graduate School of Information Sciences, Hiroshima City University, Hiroshima 731–3194, Japan
[a)] wakaba@e.kisarazu.ac.jp
[b)] wakaba@hiroshima-cu.ac.jp
[c)] s_naga@hiroshima-cu.ac.jp
[d)] inagi@hiroshima-cu.ac.jp

R_1 and R_2 be arbitrary REs, and L(R_1) and L(R_2) be regular languages corresponding to them, respectively. Then, regular expression on Σ is defined recursively as follows:

(1) A symbol $a_i \in \Sigma$ is an RE representing a regular language $\{a_i\}$.
(2) An empty character ε is an RE representing a regular language $\{\varepsilon\}$.
(3) Union $(R_1|R_2)$ is an RE representing a regular language $L(R_1) \cup L(R_2)$.
(4) Concatenation R_1R_2 is an RE representing a regular language $L(R_1R_2) = \{xy | x \in L(R_1), y \in L(R_2)\}$.
(5) Kleene closure $(R_1)^*$ is an RE representing a regular language $\{\varepsilon\} \cup L(R_1) \cup L(R_1^2) \cup \ldots$.

In addition to these RE operations, NIDSs often use Perl Compatible Regular Expression (PCRE) operators [23] to describe a pattern. We show the definitions of PCRE operators in the following.

[Definition 2]

(1) $. = a_1|a_2|\cdots|a_s$.
(2) $a^? = a|\varepsilon$.
(3) $a^+ = aa^*$.
(4) $[a_i\text{-}a_j] = a_i|a_{i+1}|\cdots|a_j$, $(1 \le i < j \le s)$.
(5) $[\hat{}\,a_i\text{-}a_j] = a_1|\cdots|a_{i-1}|a_{j+1}|\cdots|a_s$, $(1 \le i < j \le s)$.
(6) $R\{n, m\} = R^n|R^{n+1}|\cdots|R^m$, $(n < m)$.

(1) are called the wild character. (2) are called the optional character. (3) are called the Kleene plus. (4) and (5) are called the character classes. (6) is called the quantifier.

Other PCRE operators such as back reference and look ahead assertion are also used to describe a pattern in NIDSs. However, this paper does not handle these operators, just like other studies [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18].

2.2 Partial Reconfiguration

Several modern FPGAs have the partial reconfiguration (PR) function, which can modify a part of a circuit implemented in the FPGA without reconfiguring the whole circuit. Some FPGAs also support the dynamic partial reconfiguration (DPR) function, which can perform PR while the remaining logic continues to operate without interruption. In recent years, Xilinx Inc. provided PlanAhead, which is a tool to design a circuit using DPR [24].

A region in the FPGA chip area preserved for PR is called a Partial Reconfiguration Region (PRR). A module to be implemented in PRR is called a Partial Reconfiguration Module (PRM). And, the remaining region in the chip other than PRRs is called a Static Region (SR), and a module in SR is called a Static Region Module (SRM).

To realize a circuit using PR, we firstly decide the sizes of PRRs and their locations in the FPGA chip. Then, placement and routing are performed for one PRM and the SRM. Next, placement and routing for other PRMs are performed using the result of placement and routing of the SRM. Placement and routing for other PRMs are more difficult than those for a circuit without including any PRR. Therefore, some area overhead is unavoidable. Note that a circuit including more PRRs may incur a larger area overhead.

2.3 Related Works

REM has been commonly realized by software simulating deterministic finite automaton (DFA) or non-deterministic finite automaton (NFA). Since, in DFA, only one state is always active, calculation of state transition in software is simple and fast. However, a large amount of memory may be required, since the state explosion may occur in DFAs for some REs [19], [20]. Therefore, to reduce the number of states, Delayed DFA (D^2FA) [19] and Extended FA (XFA) [20] have been proposed. On the other hand, simulating an NFA in software requires a small amount of memory, since the number of states of an NFA is fewer than a corresponding DFA. However, calculation of state transition in software may be slow, since in NFA, one or more states are concurrently active. Reference [21] shows performances of software simulation for NFA and DFA combining 100 REs are about 0.4 Mbps and 20 Mbps respectively . To improve performance of software NFA simulation, Ref. [21] proposed NFA-OBDD which is 10 times faster than ordinary software simulation of an NFA, while requiring approximately the same amount of memory.

To realize fast REM, hardware implementation has been widely studied. Reference [2] has proposed a method to implement an NFA corresponding to a given pattern as a circuit on a reconfigurable device, such as an FPGA. References [3], [4], [5], [6], [7], [8], [9], [10], [11], [12] have proposed a method to improve this method. Reference [3] has improved the area efficiency and performance by optimizing the conversion from RE to NFA and from the NFA to circuit. References [7], [8] have proposed some techniques to share common prefix, infix and suffix between REs to reduce the circuit size. Reference [4] has proposed a method to optimize a circuit for FPGA with 6 input LUTs. References [5], [6] have proposed a string transition NFA to improve performance and reduce the circuit size. References [9], [10], [11] have proposed a circuit structure for quantifier and character class. Reference [12] has proposed a method which can switch a pattern dynamically according to the type of protocols in NIDSs by using dynamic partial reconfiguration. As a result, the area efficiency of the circuit is improved. These engines have to be re-designed whenever a pattern is updated. In this paper, these engines are called pattern dependent engines.

REM hardware engines which do not need re-design when a pattern is updated, have been also proposed [13], [14], [15], [16], [17], [18]. References [13], [14], [15], [16], [17] have proposed compact and fast REM engines by restricting the class of REs. References [13], [14] have proposed NFA and bit parallel NFA based REM engines, respectively. These engines handles restricted union for string as an input pattern. References [15], [16], [17] have proposed systolic algorithm based REM engines. Reference [15] has proposed a systolic algorithm based REM engine which can handle concatenation, union and Kleene closure for a character (e.q., "$a^*bc(c|de)fg$"). Reference [16] has extended the engine [15] to handle the class of REs excluding nested Kleene closure. The nested Kleene closure means a pattern R^* that contains Kleene closure in R, such as "$(a(bc)^*d)^*$." Reference [17] has extended the engine to handle quantifier directly. Reference [18] has proposed a systolic algorithm based REM engine which can handle any REs. In this

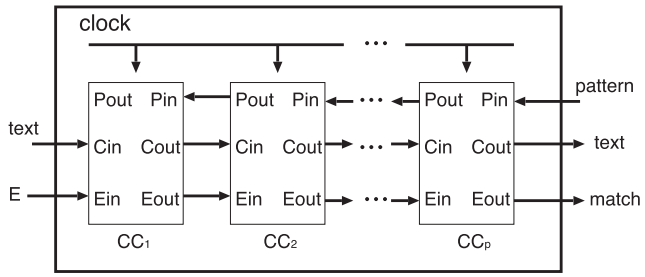

Fig. 1 The basic architecture of REM engines [15], [16], [17].

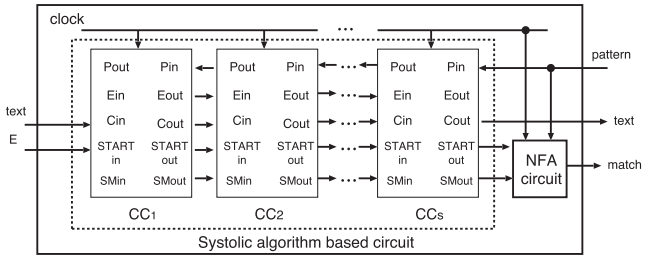

Fig. 2 The basic architecture of REM engines [18].

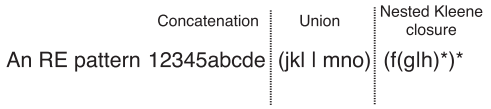

Fig. 3 An RE pattern.

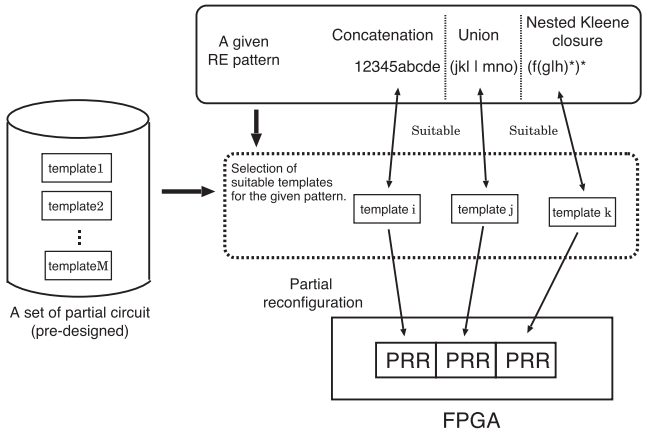

Fig. 4 The overview of the proposed method.

paper, these engines are called pattern independent engines.

In the following subsection, we explain systolic algorithm based REM engines, which is used in the proposed method.

2.4 Systolic Algorithm Based REM Engines

In this subsection, we briefly explain systolic algorithm based REM engines [15], [16], [17]. They are constructed as a one-dimensional array of simple processing units, called comparison cells (CCs), as shown in **Fig. 1**. A CC is a synchronous circuit module that performs one-character matching. A pattern is input from the rightmost CC before starting REM. A text to be retrieved is input from the leftmost CC, character by character, and one-character matching is performed in each CC in parallel and pipeline fashion. The matching result of each CC is transmitted to its right neighbor CC. A matching success signal output from the rightmost CC indicates that a substring in the text matches with the pattern. In the engines [15], [16], [17], if the engine handles a larger class of REs, each CC requires more functions and the size of a CC becomes larger.

To handle any REs, Ref. [18] has proposed a REM engine combining systolic algorithm based circuit and NFA based circuit, as shown in **Fig. 2**. This engine realizes REM by simulating a string transition NFA. In the engine, the systolic algorithm based circuit performs matching for transition characters or strings, and the NFA based circuit performs the state transition using matching result produced by the systolic based engine. While the engine can handle any REs, each CC has many functions and requires a larger circuit area than a CC in Refs. [15], [16], [17]. However, in the actual applications such as Snort [22], some functions of CCs are not always used for a pattern. This is because most of RE operators used to describe a pattern are concatenation and union, which require only simple function of CCs.

3. The Proposed Method

The best way to minimize the circuit area is to adopt the pattern dependent approach, since a dedicated circuit can be designed for a given pattern. However, whenever a pattern is updated, it needs re-design of a circuit resulting in a long updating time. To avoid time-consuming re-design while reducing the circuit area, we focus on the following two points.

(1) Sub patterns in an RE pattern may consist of a specific class of REs such as concatenation only (e.g., **Fig. 3**),

(2) A pattern independent REM engine restricting the class of REs becomes a compact circuit.

We propose a new method using partial reconfiguration (PR) by taking the above two points into account. The overview of the proposed method is shown in **Fig. 4**. In the proposed method, several partial circuits, called *templates*, each of which handles a different class of REs and performs matching for a sub pattern, are provided in advance. When an RE pattern is given, a suitable REM engine is automatically produced by combining the templates according to the pattern. Then, the engine is implemented on an FPGA using PR.

To design a REM engine using templates, we have to partition a circuit into partial circuits. In general, partition of a circuit would be a difficult problem. However, in the systolic algorithm, partition of a circuit is easy, since CCs are regularly connected as shown in Fig. 1. We can easily obtain a partial circuit by grouping successive CCs into one. Thus, we adopt the systolic algorithm for the proposed engine. The existing systolic algorithm based engine [18] has a homogeneous structure, in which all CCs have the same functions so that any RE can be handled. Thus, as mentioned before, many functions in a CC tend to be unused. To reduce the unused functions, we focus on a heterogeneous structure, in which each CC has different functions to handle a subclass of REs.

The proposed method can reduce the circuit area significantly, and produce a more compact REM engine than the existing engine [18] because of its heterogeneous structure. In addition, the proposed method can still update a pattern very quickly, since it just selects templates according to a given pattern, downloads bitfiles of the selected templates, and sets the pattern to the engine.

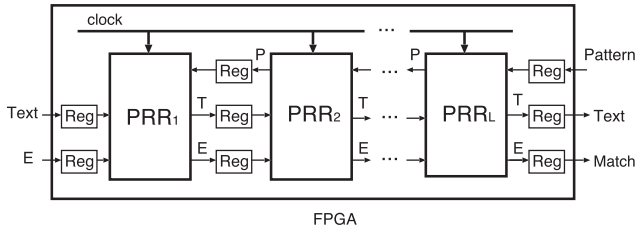

Fig. 5 The proposed circuit structure.

The following shows details of the proposed method.

3.1 The Proposed Circuit Structure

The proposed structure of the REM engine is a one-dimensional array of L PRRs in an FPGA ($L > 1$), as shown in **Fig. 5**. Each PRR realizes a partial circuit (template) obtained by grouping successive CCs explained in the next subsection. For each PRR, M kinds of templates are prepared in advance. The output signals from each PRR are text T, pattern P and matching signal E. The output signals T, P and E from each PRR are connected to its right neighbor PRR through registers. This structure can update an RE pattern without re-designing a circuit by partially reconfiguring only PRRs whose templates have to be exchanged. By designing only M kinds of templates [*1], M^L kinds of REM engines can be realized. By preparing templates for a whole circuit, we could also update a pattern quickly without using PR. But, in this case, M^L templates must be designed in advance. This is practically impossible.

3.2 Templates

In the proposed method, M kinds of templates are prepared in advance. One template is the REM engine combining systolic algorithm based circuit and NFA based circuit (Fig. 2), which handles any REs. Other templates are constructed as a one-dimensional array of CCs (Fig. 1), each of which handles a different subclass of REs. A CC which handles a larger class of REs, requires many functions and many circuit area. And, the size of PRR in which each template is implemented, is fixed. Therefore, templates have the characteristics shown in the following.

(1) A template handling a smaller class of REs can accommodate a longer sub pattern.
(2) A template handling a larger class of REs accommodates a shorter sub pattern.

In the proposed method, for a given application, preparing an appropriate set of templates is very important to reduce the circuit area. However, selecting an optimal set of templates is practically impossible since patterns to be matched with templates are not given in advance. Thus, we should adopt some heuristic approach to constructing a set of templates. Our heuristic approach is as follows. First, we define one template, T_1, which can be realized in one PRR, and can treat any regular expression which may be used in the given application of REM. Then, we arbitrarily choose a small positive integer, denoted M (>0), and arbitrarily define (M-1) subclasses of regular expressions so that those are different from each other as much as possible. Then, for each subclass of regular expressions, we define template T_i, $1 < i \leq M$, which can be realized in one PRR, and can treat its corresponding subclass of regular expressions, obtaining $\{T_1, T_2, \ldots, T_M\}$ as a set of templates to be used in REM. An example of constructing a set of templates is shown in Section 4.

[*1] The number of FPGA bit-files is ML, since a bit-file for a PRR cannot be used for other PRRs due to the restriction of FPGA design tool.

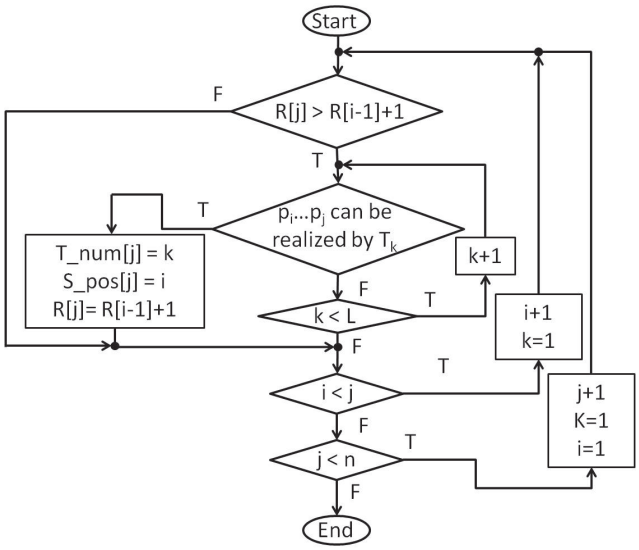

Fig. 6 The flow chart of the selection of templates.

3.3 The Selection of Templates

In the proposed method, the best combination of templates for a given pattern must be selected. The problem of selecting the best combination of templates for the given pattern is formalized as follows.

Input: An RE pattern, a set of M templates $\{T_1, T_2, \ldots, T_M\}$.
Objective function: The number of templates, R, required to handle a given pattern.
Output: A combination of templates, for which R is minimized.

To solve the optimization problem mentioned above, we use dynamic programming (DP). Let $P{=}p_1 p_2 \ldots p_n$ be a given RE pattern without RE operators such as '$*$' and '|'. $f(i)$ indicates the minimum R required by $p_1 p_2 \ldots p_i$ ($1 \leq i \leq n$), which is computed by using DP as follows.

$f(0) = 0$,
$f(i) = min(f(0) + D(1,i), f(1) + D(2,i), f(2) + D(3,i), \ldots, f(i-1) + D(i,i)), 1 \leq i \leq n$.

$$D(x,y) = \begin{cases} 1 & \text{(some template can realize } p_x \ldots p_y\text{)} \\ \infty & \text{(no template can realize } p_x \ldots p_y\text{)} \end{cases}$$

The $D(x,y)$ indicates whether $p_x \ldots p_y$ can be realized by a template or not by using a greedy algorithm. Each p_i ($x \leq i \leq y$) is realized by a CC in a template. If p_i is not included in the class of REs which a CC can handle, then ε is set to the CC, and realizing p_i at its right neighbor CC is tried. When we run short of CCs in the template, we find that the template cannot realize $p_x \ldots p_y$.

We show the flow chart of the proposed DP in **Fig. 6**. In the flow chart, three variables i, j, k and three arrays $R[0 \ldots n], T\_num[0 \ldots n], S\_pos[0 \ldots n]$ are used. $R[j]$ stores the minimum number of templates needed to realize $p_1 \ldots p_j$. If $p_i \ldots p_j$ is realized by a template T_k, $T\_num[j]$ and $S\_pos[j]$ store the template number k and the index of the head character i, re-

Table 1 The comparison cell (CC) used in templates.

Type	The class of REs	Pattern examples	Size
1	Concatenation for a character, epsilon, the wild character and the optional character	$abc, \varepsilon, x.z^?$	1.0
2	Kleene closure for a character or a class character	$[a\text{-}z]b^+c, [\hat{}a\text{-}z]^*bc.^?$	1.4
3	Union	$([a\text{-}k]b\|c)(d\|e^*f), a(b\|(c\|d)e).^*$	1.8
4	Quantifier for a character or a class character (The number of repetitions is up to 10).	$a\{10\}bc, (ab\|c[k\text{-}z]\{5, 8\})$	2.3
5	Quantifier for a character or a class character (The number of repetitions is up to 30).	$a\{30\}bc, (ab\|c[k\text{-}z]\{4, 15\})$	3.0

spectively. As initial values, i=j=k=1, $R[0]$=0, $R[1 \ldots n]$=∞ are given.

To obtain the value of $f(i)$, each $D(j, i)$ for $1 \leq j \leq i$ has to be computed sequentially. In the flow chart, to avoid time-consuming computation of $D(j, i)$ as much as possible, we estimate the value of $D(j, i)$ as its lower bound (i.e., 1) before computing $D(j, i)$. If the sum of $f(j-1)$ and the estimated value (i.e., $f(j-1)+1$) is larger than the already computed value for $f(i)$, then computation of $D(j, i)$ is skipped.

The minimum number of templates required by P is stored in $R[n]$. The best combination of templates can be obtained by trace-back using T_num and S_pos.

[The time complexity of selecting templates]

To select templates, two processes are required. In the following, normal characters (i.e., characters in alphabet Σ) in a pattern are called pattern characters. First, we convert a given pattern into pattern characters with their information about the RE class. Next, we calculate the best combination of templates.

Let P_{op}=$q_1 q_2 \ldots q_v$ be a given pattern. Let P=$p_1 p_2 \ldots p_n$ be pattern characters ($n \leq v$ and $p_i \in \Sigma$). For each pattern character p_i, the RE class information is associated to describe which RE class is required to handle it. Let $T_1, \ldots, T_M$ be the given templates. Let L_s be the number of CCs in $T_s \in \{T_1, T_2, \ldots T_M\}$, and k be the maximum value among $L_1, L_2, \ldots$ and L_M.

First, we discuss the time complexity of the conversion of a pattern. This process determines the RE class of each pattern character based on the class of REs which CCs in templates have, as shown in **Table 1**. For example, given "$(a|b)c$," the RE class for 'a', 'b' and 'c' are 3, 3 and 1, respectively. In the calculation, first, a pattern is converted to the reverse Polish notation by the syntactic analysis, then the RE class for each pattern character is determined by reading from the end of the reversed pattern to the beginning of the reversed pattern. For example, in "$(a|b)c$" and "$(ab|c)$," the RE class of each pattern character is decided based on an effective range of application of each RE operator and its precedence as shown in **Fig. 7**, respectively. The time complexity of syntactic analysis and setting RE classes are $O(v)$, respectively, since an RE pattern is read from the beginning of the pattern to the end of the pattern, and from the end of the reversed pattern to the beginning of the reversed pattern. Details of these processes are omitted.

Next we discuss the time complexity of calculating templates. First, we discuss the time complexity of calculating the value of $D(j, i)$. To calculate the value of $D(j, i)$, for each template $T_s \in \{T_1, T_2, \ldots T_M\}$, we have to check whether $\mathrm{CC}_1 \ldots \mathrm{CC}_{L_s}$ can realize $p_j p_{j+1} \ldots p_i$ or not. The time complexity of a process to check whether CC_r ($1 \leq r \leq L_s$) can realize p_h ($j \leq h \leq i$) or not, can be considered as $O(1)$, since the process is a simple comparison between the class of REs of CC_r and the class of REs of p_h. Therefore, the time complexity of the process to check whether $p_j p_{j+1} \ldots p_i$ can be realized by $\mathrm{CC}_1 \ldots \mathrm{CC}_{L_s}$ or not, is $O(L_s)$ in the worst case, since it is determined that the template cannot realize $p_j \ldots p_i$ when we run short of CCs of the template. Thus, the time complexity of calculating the value of $D(j, i)$ is $O(u \times M)$, where $u = (L_1+L_2+\ldots+L_M)/M$. Please note that $u \leq k$.

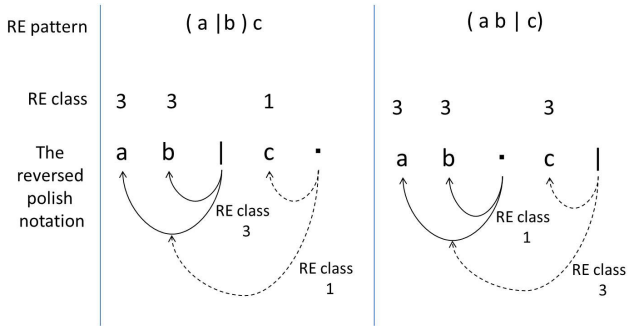

Fig. 7 A process to set an RE class to each pattern character.

Next, we discuss the time complexity of calculating the value of $f(i)$. To calculate the value of $f(i)$, the calculations of the value of $D(j, i)$ ($1 \leq j \leq i$) are conducted. Therefore, the time complexity of calculating the value of $f(i)$ is $O(u \times M \times i)$.

Finally, we discuss the time complexity of selecting templates for a given pattern P=$p_1 p_2 \ldots p_n$. To select templates for the pattern, the calculations of $f(1)$ ($O(u \times M \times 1)$), $f(2)$ ($O(u \times M \times 2)$), $\ldots$ and $f(n)$ ($O(u \times M \times n)$) are required. Therefore, the time complexity of selecting templates for the pattern is $O(u \times M \times n^2)$ ($= \Sigma_{i=1}^{n} O(u \times M \times i)$).

The problem of selecting templates can be solved more efficiently by calculating the value of $D(j, i)$ using the value of $D(j, i\text{-}1)$, which was previously calculated. It can improve the practical computation time, and the C program used in the experiments discussed in the next section adopted this method.

4. Experimental Evaluation

4.1 Experimental Environment

For the design of the proposed circuit structure and templates, we use Xilinx ISE13.1 and Xilinx PlanAhead13.1. The target FPGA is Virtex-6 (xc6vlx240tff1156-11) which has 37,680 Slices. The C program for the selection of templates was run on a PC, in which, OS is Ubuntu12.04 (on VMware), CPU is Intel(R) Core i7-3770K 3.50 GHz, Memory is 4 GB. And, we use 2,052 patterns excluding extended RE patterns with back reference and look ahead assertion, in Snort rule v2.9 [22].

4.2 Prepared Template

Types of CCs used in templates are shown in Table 1. Table 1

shows a class of REs which each CC can handle, and the relative circuit size of each CC against type 1. Details of CCs are given as follows.

(1) Type i can also handle a class of REs which type j ($j < i$) can handle.
(2) Type 4 or 5 handles quantifiers with up to 10 or 30 repetitions of a character or a class character, respectively.

Templates are designed by combining the above five types of CC. In this paper, 18 kinds of templates are prepared (M=18). One template is the REM engine which handles any REs. 5 kinds of templates consist of only one of the 5 types. 12 ($=_4P_2$) kinds of those consist of two different types of CCs, which are chosen among types 1 to 4 (**Fig. 8**).

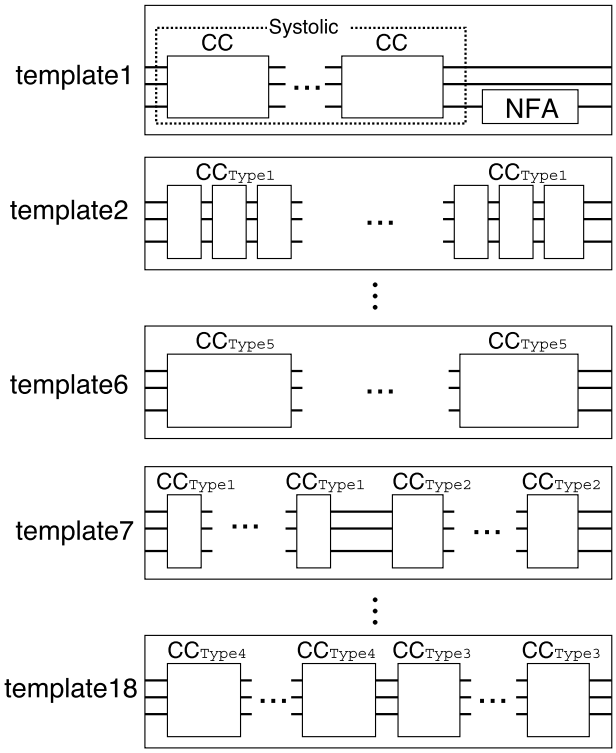

Fig. 8 Kinds of prepared templates.

4.3 Area Evaluation

First, to evaluate the circuit area of the proposed engines based on various numbers of PRRs, we compare them with the existing REM engine [18]. 18 kinds of templates are prepared as shown in Section 4.2. A histogram in **Fig. 9** shows the maximum number of Slices required to realize each of 2,052 patterns in the proposed engines and the existing engine. In Fig. 9, L denotes the number of PRRs. The proposed engines (L= 3 and 6) are implemented using half Slices compared to Ref. [18], and the numbers of Slices of the engines (L=9 and 12) are less than that of the engines (L=3 and 6). However, in the engines (L=15 and 18), the number of Slices was larger compared to the engines (L=9 and 12). In the proposed method, the engine with more PRRs can reduce more circuit areas by reducing more unused functions, since for a given pattern, the templates can be selected more flexibly. However, the engine with more PRRs has a larger area overhead of PR as described in Section 2.2. In the engines (L=15 and 18), because the area overhead is greater than the reduction effect of circuit area by flexibly selecting templates, the total circuit area becomes larger. Therefore, the proposed method using PR can improve the area efficiency unless the number of PRRs is increased extremely. The proposed REM engine (L=12) performs matching with 40% circuit area of the existing engine [18].

4.4 Performance Evaluation

We evaluate the performance of the proposed REM engine. A line chart in Fig. 9 shows the maximum clock frequency of the proposed engines and the existing engine [18]. The maximum clock frequency of the existing engine is 186 MHz. On the other hand, those of the proposed engines are from 114 MHz to 163 MHz. This degradation of the performance may be explained by an overhead of PR. Although the clock frequency was degraded, the proposed REM engine (L=12), which is the most compact one, can be still used in NIDSs for the Gigabit Ethernet, since the throughput of the proposed REM engine (L=12) achieves 1.1 (= 8×141 MHz) Gbps.

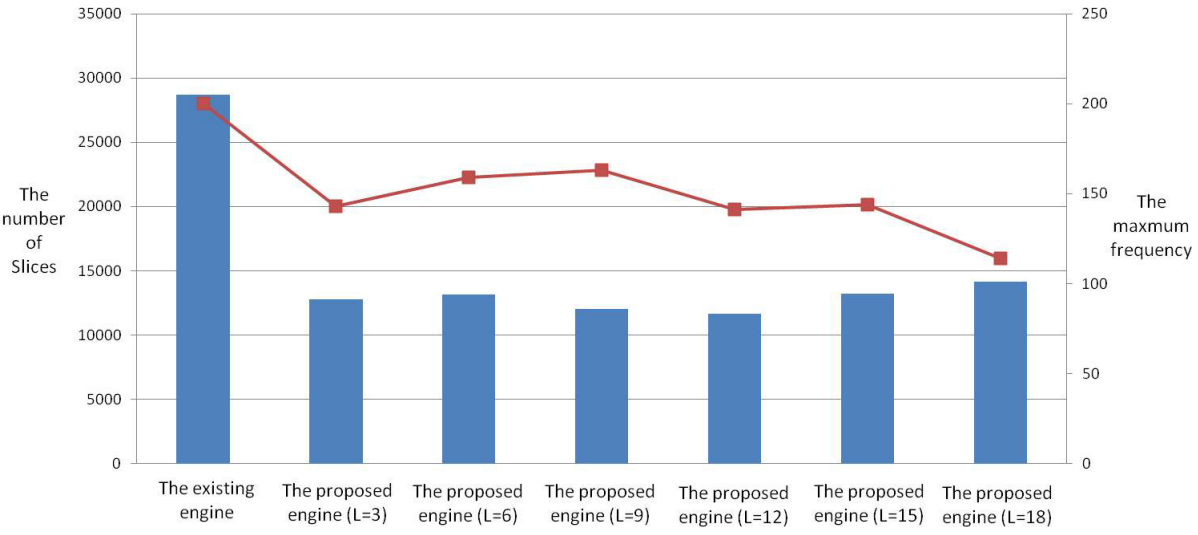

Fig. 9 The circuit areas and performances of the existing engine [18] and the proposed engine.

Table 2 The computation time of selecting templates for various numbers of PRRs.

The number of PRRs (L)	3		6		9	
	Normal	Improvement	Normal	Improvement	Normal	Improvement
The minimum (s)	2.98×10^{-4}	2.85×10^{-4}	2.97×10^{-4}	2.88×10^{-4}	2.97×10^{-4}	2.86×10^{-4}
The maximum (s)	8.39	2.37	6.82	3.83	6.42	4.61
The average (s)	4.10×10^{-2}	0.34×10^{-2}	3.28×10^{-2}	0.65×10^{-2}	2.97×10^{-2}	0.95×10^{-2}
The maximum improvement rate	-	1%	-	2%	-	2%
The average improvement rate	-	58%	-	54%	-	53%

The number of PRRs (L)	12		15		18	
	Normal	Improvement	Normal	Improvement	Normal	Improvement
The minimum (s)	2.94×10^{-4}	2.76×10^{-4}	2.89×10^{-4}	2.76×10^{-4}	2.88×10^{-4}	2.87×10^{-4}
The maximum (s)	6.16	4.89	5.97	4.94	5.81	5.24
The average (s)	2.73×10^{-2}	1.10×10^{-2}	2.56×10^{-2}	1.15×10^{-2}	2.41×10^{-2}	1.32×10^{-2}
The maximum improvement rate	-	2%	-	2%	-	3%
The average improvement rate	-	48%	-	48%	-	45%

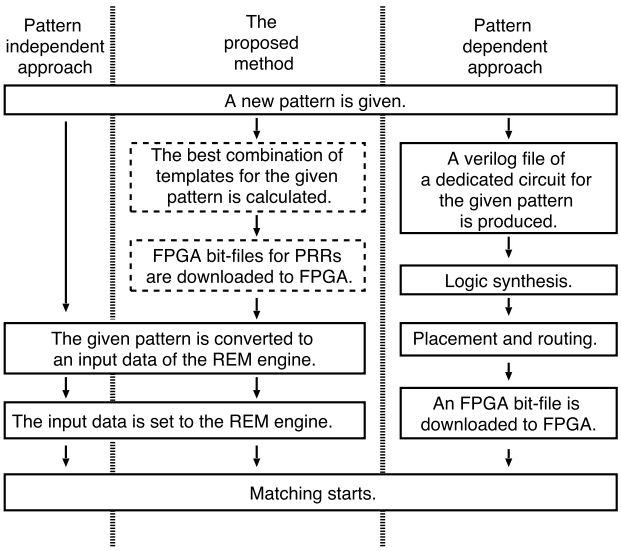

Fig. 10 The flow of preparation to perform matching in each approach.

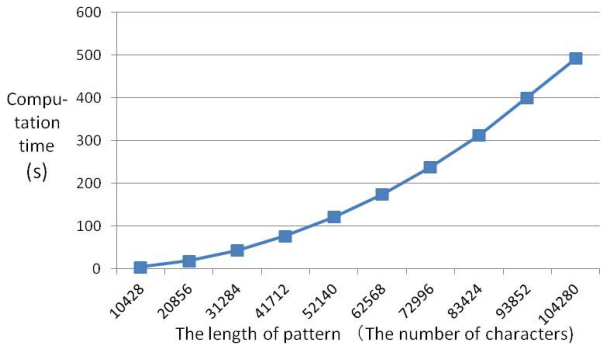

Fig. 11 The computation time of selecting templates for patterns with various lengths.

4.5 Evaluation of Pattern Updating Time

In this subsection, we evaluate the pattern updating time. **Figure 10** shows the flow of preparation to perform matching in pattern dependent approach, pattern independent approach and the proposed method. For updating a pattern, the proposed method requires to select templates and download FPGA bit-files, in addition to the pattern updating process of the existing method [18], which takes about 2 seconds. First, we evaluate the time of selecting templates. **Table 2** shows the minimum, the maximum and the average computation time of selecting templates for each pattern in 2,052 patterns. In Table 2, *Improvement* means the case that the C program estimates the value of $D(j,i)$ as its lower bound (i.e. 1) before computing $D(j,i)$ described in Section 3.3, and *Normal* means the case that the C program does not estimate the value of $D(j,i)$. The computation time of the *Improvement* C program is at most 1% and on average half compared to that of the *Normal* C program. Table 2 shows that our proposed method requires at most 5.24 seconds for selecting templates. On the other hand, the existing pattern dependent approach [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12] requires more than 5 minutes due to re-design of a dedicated circuit for a pattern.

Next, we describe the relation between the length (the number of characters) of a pattern and the computation time of selecting templates. **Figure 11** shows the computation time for patterns, each of which has N times characters of 10,428 characters (N=1,2,...,10), which is the maximum length of a pattern in Snort rule v2.9. In this experiment, a pattern which has N times characters is produced by concatenation of a pattern which has 10,428 characters N times. Figure 11 shows that the computation time of the proposed method increases quadratically with the length of a pattern. The proposed method can produce a REM engine for a pattern of length 104,280 in 8 minutes. Therefore, the proposed method can select the best combination of templates for a very long pattern quickly.

Next, we describe the circuit area and the computation time of selecting templates when the number of templates was changed. **Figure 12** shows the average circuit area and the average computation time for 2,052 patterns in the proposed engines. In Fig. 12, 6, 12 and 18 indicate the number of templates used in the experiments. The case of 6 templates consists of template 1, 4, 7, 10, 13 and 16. The case of 12 templates consists of template 1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16 and 17. The case of 18 templates consists of all templates. Figure 12 indicates that the circuit area decreases linearly and the computation time increases linearly as the number of templates increases.

Finally, we evaluate the download time of FPGA bit-files. The proposed method uses JTAG to download bit-files to an FPGA. The download speed with JTAG is 66 Mbps. The maximum size

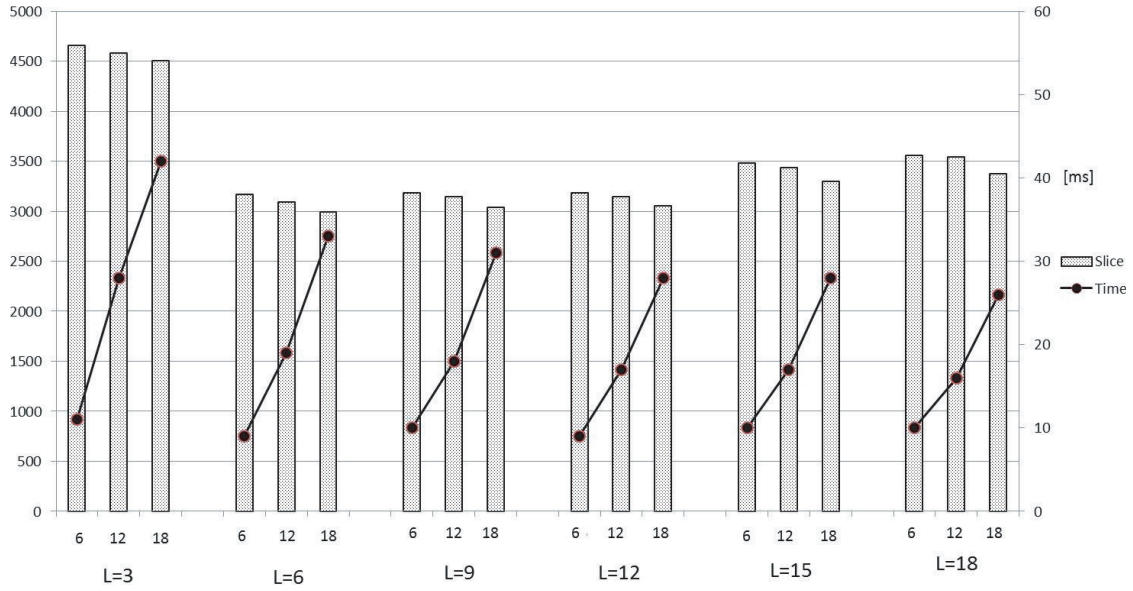

Fig. 12 The circuit area and the computation time of selecting templates for the number of templates.

of an FPGA bit-file for each PRR is 1.5 M bits. Therefore, the download time of FPGA bit-files is 1 second or less. Thus, the pattern updating time of the proposed method is at most 9 (=5.24 (template selection)+1(download)+2(pattern set)) seconds, that is comparable to Ref. [18].

5. Conclusions

This paper proposes a method using partial reconfiguration to realize a compact REM engine, which can update a pattern quickly. In the proposed method, a set of partial circuits, each of which handles a different class of REs, are provided in advance. Given an RE pattern, a compact matching engine is automatically produced by combining the partial circuits according to the given pattern. Experimental results show that the proposed method reduces 60% circuit area compared to the existing approach [18] without increasing the pattern updating time significantly.

As described in the Introduction, for building REM engines using FPGAs, two approaches have been widely known; one is the pattern independent approach and the other the pattern dependent (also known as "instance-specific") approach. The proposed REM engine can be considered as the third approach. As far as we know, this is the first proposal of effectively utilizing the partial reconfiguration function of FPGAs for constructing REM engines. Similar approaches based on partial reconfiguration may be possible for solving combinatorial problems using FPGAs.

As future works, the proposed method do not guarantee area reduction capability for future patterns, since the proposed method uses a heuristic approach to prepare a set of templates. Preparing an appropriate set of templates is very important to reduce the circuit area depending on applications and to guarantee area reduction capability for future patterns. Thus, we will study how to make the optimal set of templates.

Another future works include the improvement of the performance of the proposed REM engine. In the tool which we use to make templates and static circuit (SRM), we have to decide the sizes of PRRs and their locations in the FPGA chip in advance described in Section 2.2. In our implementation of the proposed REM engine, we decided the sizes of PRRs and their locations without considering the characteristics of design for partial reconfiguration such as clock region [24]. Therefore, we think that the performance of the proposed REM engine can be improved by considering the characteristics of design for partial reconfiguration.

References

[1] Hopcroft, J.E., Motwani, R. and Ullman, J.D.: *Introduction to Automata Theory, Languages, and Computation* (2nd Edition), Addison Wesley, USA (Nov. 2000).
[2] Bispo, J., Sourdis, I., Cardoso, J.M.P. and Vassiliadis, S.: Regular expression matching for reconfigurable packet inspection, *Proc. 2006 IEEE International Conference on Field Programmable Technology*, pp.119–126 (Dec. 2006).
[3] Ganegedara, T., Yang, Y.E. and Prasanna, V.K.: Automation Framework for Large-Scale Regular Expression Matching on FPGA, *Proc. 2010 IEEE International Conference on Field Programmable Logic and Applications*, pp.50–55 (Sep. 2010).
[4] SangKyun, Y. and KyuHee, L.: Optimization of Regular Expression Pattern Matching Circuit Using At-Most Two-Hot Encoding on FPGA, *Proc. 2010 IEEE International Conference on Field Programmable Logic and Applications*, pp.40–43 (Sep. 2010).
[5] Yamagaki, N., Sidhu, R. and Kamiya, S.: High-Speed Regular Expression Matching Engine Using Multi-Character NFA, *Proc. 2008 IEEE International Conference on Field Programmable Logic and Applications*, pp.131–136 (Sep. 2008).
[6] Nakahara, H., Sasao, T. and Matsuura, M.: A regular expression matching circuit based on a decomposed automaton, *Proc. 7th International Conference on Reconfigurable Computing: Architectures, tools and applications*, pp.16–28 (Mar. 2011).
[7] Clark, C.R. and Schimmel, D.E.: Scalable Parallel Pattern Matching on High-Speed Networks, *Proc. 12th IEEE Symposium on Field Programmable Custom Computing Machines*, pp.249–257 (Apr. 2004).
[8] Hieu, T.T., Thinh, T.N., Vu, T.H. and Tomiyama, S.: Optimization of Regular Expression processing circuit for NIDS on FPGA, *Proc. 2011 IEEE Second International Conference on Networking and Computing*, pp.105–112 (Nov. 2004).
[9] Sidhu, R. and Prasanna, V.K.: Fast Regular Expression Matching Using FPGAs, *Proc. 2001 IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp.227–238 (May 2001).
[10] Clark, C.R. and Schimmel, D.E.: Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns, *Proc. 2003 IEEE International Conference on Field Programmable Logic and Applications*, pp.956–959 (Sep. 2003).
[11] Long, L.H., Hieu, T.T., Tai, V.T., Hung, N.H., Thinh, T.N. and Anh

Vu, D.D.: Enhanced FPGA-based architecture for regular expression matching in NIDS, *Proc. 2010 IEEE International Conference on Electrical Engineering/Electronics Computer Telecommunications and Information Technology*, pp.666–670 (May 2010).
[12] Salvatore, P., Claudio, G., Enrico, N., Simone, T. and Giuseppe, B.: Exploiting Dynamic Reconfiguration for FPGA Based Network Intrusion Detection Systems, *Proc. 2010 IEEE International Conference on Field Programmable Logic and Applications*, pp.10–14 (Sep. 2010).
[13] Divyasree, J., Rajashekar, H. and Varghese, K.: Dynamically reconfigurable regular expression matching architecture, *Proc. International Conference on Application-Specific Systems, Architectures and Processors*, pp.120–125 (July 2008).
[14] Kaneta, Y., Yoshizawa, S., Minato, S., Arimura, H. and Miyanaga, Y.: Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching, *Proc. 2010 IEEE International Conference on Field Programmable Technology*, pp.21–28 (Dec. 2010).
[15] Kawanaka, Y., Wakabayashi, S. and Nagayama, S.: A Systolic Regular Expression Pattern Matching Engine and its Application to Network Intrusion Detection, *Proc. 2008 IEEE International Conference on Field Programmable Technology*, pp.297–300 (Dec. 2008).
[16] Kawanaka, Y., Wakabayashi, S. and Nagayama, S.: A Systolic String Matching Algorithm for High-Speed Recognition of a Restricted Regular, *Proc. 2009 Engineering of Reconfigurable System and Algorithms*, pp.151–157 (July 2009).
[17] Wakaba, Y., Inagi, M., Wakabayashi, S. and Nagayama, S.: An Extension of Systolic Regular Expression Matching Hardware for Handling Iteration of Strings Using Quantifiers, *Proc. the 16th Workshop on Synthesis and System Integration of Mixed Information technologies*, pp.412–417 (Oct. 2010).
[18] Wakaba, Y., Inagi, M., Wakabayashi, S. and Nagayama, S.: An Efficient Hardware Matching Engine for Regular Expression with Nested Kleene Operators, *Proc. 2011 IEEE International Conference on Field Programmable Logic and Applications*, pp.157–161 (Sep. 2011).
[19] Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P. and Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection, *Proc. SIGCOMM'06*, pp.339–350 (Sep. 2006).
[20] Smith, R., Estan, C., Jha, S. and Kong, S.: Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata, *Proc. SIGCOMM'08*, pp.207–218 (Aug. 2008).
[21] Yang, L., Karim, R., Ganapathy, V. and Smith, R.: Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams, *Proc. RAID 2010*, Vol.6307 of Lecture Notes in *Computer Science* (*LNCS*), pp.58–87 (Sep. 2010).
[22] Sourcefire, Inc.: SNORT Network Intrusion Detection System, available from ⟨http://www.snort.org/⟩.
[23] Philip Hazel: PCRE — Perl Compatible Regular Expressions, available from ⟨http://www.pcre.org/⟩.
[24] Xilinx Inc.: Partial Reconfiguration User Guide, available from ⟨http://japan.xilinx.com/support/documentation/sw_manuals_j/xilinx14_6/ug702.pdf⟩.

Yoichi Wakaba received his B.S., M.E. Ph.D degrees from Hiroshima City University, Hiroshima, Japan, in 2009, 2011 and 2014, respectively. He is now an Assistant Professor in Faculty of Electrical and Electronic Engineering, Kisarazu National College of Technology. His research interest includes regular expression matching and combinatorial problems.

Shin'ichi Wakabayashi received his M.E. and Ph.D. degrees in systems engineering from Hiroshima University in 1979, 1981, and 1984, respectively. He was a researcher at Tokyo Research Laboratory, IBM Japan Ltd., in 1984–1988. From 1988 to 2003, he was an Associate Professor in Faculty of Engineering, Hiroshima University. Since 2003, he has been a Professor in Faculty of Information Sciences, Hiroshima University. His research interests include VLSI design, VLSI CAD and combinatorial optimization.

Shinobu Nagayama received his B.S. and M.E. degrees from Meiji University, Kanagawa, Japan, in 2000 and 2002, respectively, and his Ph.D. degree in computer science from Kyushu Institute of Technology, Japan, in 2004. He is now an Associate Professor at Hiroshima City University, Japan. He received the Outstanding Contribution Paper Awards from the IEEE Computer Society Technical Committee on Multiple-Valued Logic (MVL-TC) in 2005 and 2013 for papers presented at the International Symposium on Multiple-Valued Logic in 2004 and 2012, respectively, the Young Author Award from the IEEE Computer Society Japan Chapter in 2009, and the Outstanding Paper Award from the Information Processing Society of Japan (IPS) in 2010 for a paper presented at the IPSJ Transactions on System LSI Design Methodology. His research interest includes decision diagrams, analysis of multi-state systems, logic design for numeric function generators, regular expression matching, and multiple-valued logic.

Masato Inagi received his B.E., M.E. degrees in computer science, and his Ph.D. degree in engineering from Tokyo Institute of Technology in 2000, 2002, and 2008, respectively. He was a Researcher at The University of Kitakyushu from 2005 to 2008. He has been a Research Associate at the logic circuits and systems lab., Hiroshima City University, since 2008. His research interest includes combinatorial algorithms for VLSI design automation and reconfigurable systems.

(Recommended by Associate Editor: *Akio Hirata*)