APSIPA ASC 2011 Xi'an

High-speed String and Regular Expression Matching on FPGA

Yusaku Kaneta*, Shingo Yoshizawa*, Shin-ichi Minato*†, and Hiroki Arimura*
* Graduate School of Information Science and Technology, Hokkaido University,
N14 W9, Sapporo 060-0814, Japan
E-mail: {y-kaneta, minato, arim}@ist.hokudai.ac.jp, yosizawa@csm.ist.hokudai.ac.jp
Tel/Fax: +81-11-706-7680/+81-11-706-7680
† ERATO MINATO Discrete Structure Manipulation System Project, Japan Science and Technology Agency,
N14 W9, Sapporo 060-0814, Japan

***Abstract*—In recent FPGA researches, there has been much attention to dynamically reconfigurable algorithms that can modify their configuration on-the-fly. In this paper, we report recent progress on dynamically reconfigurable hardwares on FPGA for high-speed string and regular expression matching, which have been developed by our group since 2008. In particular, we describe the architecture, algorithms, and implementations of our pattern matching hardwares. We propose a pattern matching architecture, called dynamically reconfigurable bit-parallel NFA architecture which is the first dynamically reconfigurable hardware based on bit-parallel simulation of non-deterministic finite automata (NFA). This architecture enables fast dynamic reconfiguration of the patterns as well as high-throughput pattern matching for complex subclasses of regular expressions such as extended patterns, network expressions, and extended network expressions. In this approach, the information of an input NFA is compactly encoded in bit-masks stored in a collection of registers and block RAMs. Then, the NFA is efficiently simulated by a fixed circuitry using a combination of bit- and arithmetic-operations on these bit-masks consuming one input letter per clock. Experimental results show that our architecture has advantages over the previously proposed architectures in the terms of reconfiguration and running times.**

I. Introduction

A. Backgrounds

By rapid growth of network and sensor technologies, massive data of new types, called *data streams*, and related applications have emerged in various fields including networks and data engineering. ESP (event stream processing) [1] and NIDS (network intrusion detection system) [3] are example applications of data stream processing. Consequently, efficient data stream processing technologies have been extensively studied in theory and practice.

The *large-scale pattern matching problem* is one of the most important problems in data stream processing, where a pattern matching system has to work with a *large number* (e.g., thousands) of *complex patterns* (e.g., regular expressions) against *high-speed data streams* (e.g., of several Gbps). These problems are, however, quite CPU-intensive tasks and it is difficult for software on a CPU to efficiently process massive data streams real time in wire-speed. Therefore, researches on large-scale pattern matching on reconfigurable hardwares such as FPGA have attracted much attention recently ([3], [4], [5], [7], [10], [11], [13], [14]).

B. Our research goal

A recent research trend to large-scale regular expression matching hardwares is to simulate finite state automata for a given class of regular expressions on a specially designed hardware ([3], [4], [5], [10], [11], [13], [14]). Then, this approach is further classified into the static compilation approach and the dynamic reconfiguration approach.

In the *static compilation approach* ([10], [11], [13], [14]), a set of input regular expressions is transformed into either deterministic finite automata (DFA) or non-deterministic finite automata (NFA) [9], and then statically compiled into wired logic on FPGA. However, the static compilation approach has a drawback that modification of regular expressions is too expensive to be done frequently.

In the *dynamic reconfiguration approach* ([3], [4], [5]), a universal control logic is statically compiled into FPGA beforehand, a description of regular expressions is loaded to the FPGA as data in the preprocessing phase, and then simulated in the run-time phase. This approach is attractive in real world applications such as EPS and NIDS where reconfiguration of input patterns frequently occurs. However, since classes of patterns that can be dealt with in this approach are still limited, our goal is to design dynamically reconfigurable hardwares that efficiently run for wider classes of regular expressions.

C. Main results

In this paper, we report recent progress on dynamically reconfigurable hardwares on FPGA for high-speed string and regular expression matching. We propose a novel pattern matching architecture, called *dynamically reconfigurable bit-parallel NFA architecture*. We show the top-level of our architecture in Fig. 1.

A *pattern matching module* (PMM, for short) is a core of our architecture, which is responsible for NFA-simulation of an input pattern. The key to the construction of a PMM is the use of *bit-parallel NFA-simulation method* developed in string matching communities since 1990 ([2], [9], [12]). In this method, the information of an NFA is compactly encoded

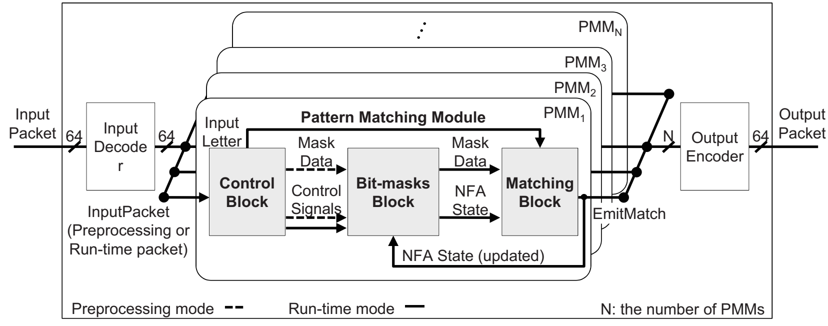

Fig. 1. The top-level of our pattern matching architecture.

in bit-masks. Then, the NFA is efficiently simulated by a fixed control logic using a combination of bit- and arithmetic-operations on these bit-masks consuming one input letter per clock [9].

An advantage of our architecture is the worst-case performance guaranteed by the design unlike the DFA-based architecture with micro controller [3]. Another advantage is the potential extensibility to more general pattern classes. For example, Kaneta *et al.* [6] recently extended the Extended SHIFT-AND method [9], used in this paper, to more general classes of network and regular expressions allowing union and the Kleene-star. Such method can be incorporated into our architecture by extending the construction of bit-masks and a circuitry described in this paper.

This paper is organized as follows. In Section II, we give basic definitions. In Section III, we propose our architecture, and in Section IV, we give the detailed description of a pattern matching module for extended patterns. In Section V, we give experimental results. In Section VI, we show an extension to more complex subclasses of regular expressions, and in Section VII, we conclude.

II. Preliminary

A. Regular expression matching

Let $\mathbf{N} = \{0, 1, 2, \ldots\}$ be the set of all non-negative integers, and $\Sigma = \{a, b, \ldots\}$ be a finite alphabet of *letters*. A *string* on Σ is a sequence $S = s_1 \cdots s_n$ of letters, where $S[i] = s_i \in \Sigma$ for every $1 \leq i \leq n$. We denote by $S[i..j]$ the substring $s_i \cdots s_j$ for every $i \leq j$, and by ε the *empty string*. If $i > j$, we define $S[i..j] = \varepsilon$. For a set $S \subseteq \Sigma^*$ of strings, we denote by $|S|$ the cardinality and $||S|| = \Sigma_{s \in S}|s|$ the total size of S. We denote by Σ^* the set of all strings on Σ. For a letter $a \in \Sigma$ and an integer $i \in \mathbf{N}$, we define by a^i the string consisting of i consecutive a.

Let REG be the class of regular expressions on Σ. More precisely, a regular expression R is either a letter $a \in \Sigma$, concatenation $R = R_1 \cdot R_2$, union $R = (R_1|R_2)$, and the Kleene-star $R = (R_1)^*$, where R_1 and R_2 are regular expressions [9]. For a regular expression $R \in \mathsf{REG}$, we denote by $L(R) \subseteq \Sigma^*$ its language. Let $T = t_1 \cdots t_n \in \Sigma^*$ be an *input text* of length $n \geq 0$, where $t_i \in \Sigma$ $(1 \leq i \leq n)$. A *pattern* is a regular expression on Σ. We say a regular expression $R = r_1 \cdots r_m \in \mathsf{REG}$ occurs at the end position j in T, if $T[i..j] = t_i \cdots t_j \in L(R)$. Our problem is stated as follows.

Definition 1. *The multiple pattern matching problem* for a subclass $\mathcal{C} \subseteq \mathsf{REG}$ of regular expressions is defined as follows. An input is an input *pattern set* $\mathcal{P} = \{\ (i, R_i)\,|\, i = 1, \ldots, N\ \} \subseteq \mathcal{C}$ $(N \geq 1)$, where for every $i = 1, \ldots, N$, R_i is a pattern and i is an integer, called an *index*. Then, the task is, given a stream $T = t_1 t_2 \cdots t_p \cdots$ $(p \geq 1)$ of input letters, to output the pairs (i, p) such that $i = 1, \ldots, N$ is the index and p is an end position of the R_i in T for all $p = 1, 2, \ldots$.

B. Target pattern classes

The target subclasses of regular expressions that our architecture deals with are the classes of *extended patterns* (EXT), *network expressions* (NET), and *extended network expressions* (EXNET) defined as follows [9]. In what follows, $\equiv$ means the notational equivalence.

An extended pattern (over Σ) in EXT is a regular expression in linear form, that is, $R = r_1 \cdots r_m$ $(m \geq 0)$, where for every $1 \leq i \leq m$, r_i is one of the following forms: (i) letters $a \in \Sigma$, (ii) wildcards $. \equiv \Sigma$, (iii) classes of letters $\alpha = [ab \cdots]$, (iv) optional letters $\alpha? \equiv (\alpha|\varepsilon)$, (v) bounded repeats $\alpha\{x, y\} \equiv (\alpha?)^{y-x}\alpha^x$, and (vi) unbounded repeats α^* and $\alpha^+ \equiv (\alpha\alpha^*)$, where $\alpha \subseteq \Sigma$. Note that a letter $a \in \Sigma$ and a wildcard '.' are classes of letters. If a component r_i is one of the forms $\alpha?$, $\alpha\{x, y\}$, α^*, and α^+, then α is called the *matrix* of r_i. We say that R is an *exact string pattern* (also called a *string pattern*), denoted by STR, if every component r_i of extended pattern $R = r_1 \cdots r_m$ is a letter in Σ.

A network expression (over strings) in NET is a regular expression without the Kleene-star, that is, a regular expression obtained from strings, concatenation, and union. An extended network expression in EXNET is a network expression over extended patterns.

Example 1. *We show examples of string patterns (*STR*), extended patterns (*EXT*), network expressions (*NET*), and extended network expressions (*EXNET*):*

- $R_1 = \mathtt{ABABC} \in \mathsf{STR}$.
- $R_2 = ([\mathtt{AB}]+)(\mathtt{B}.\{1,3\})([\mathtt{BC}]?)(.^*)\mathtt{C} \in \mathsf{EXT}$.
- $R_3 = \mathtt{A}(\mathtt{AB}|\mathtt{B})(\mathtt{B}|\mathtt{AB})\mathtt{C} \in \mathsf{NET}$.

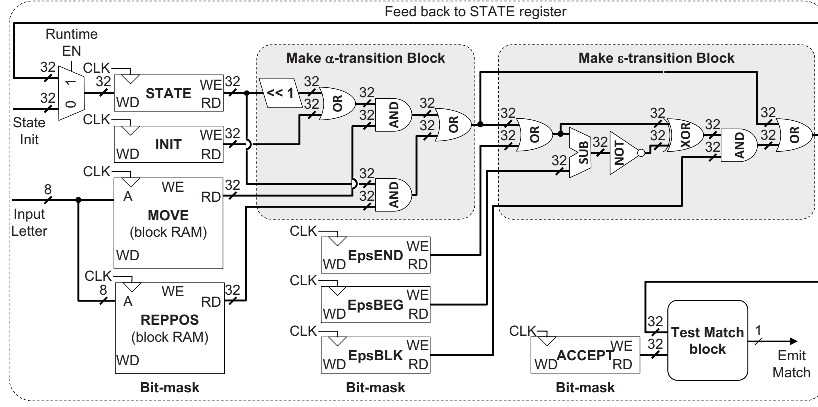

Fig. 2. The circuit of a pattern matching module for extended patterns.

- $R_4 = \mathtt{A(AB|B?)(B?.*|AB)C} \in \mathsf{EXNET}$.

III. Proposed Architecture

In this section, we present our dynamically reconfigurable bit-parallel NFA architecture based on bit-parallel NFA-simulation method ([2], [9], [12]). For the details, please consult our papers ([6], [7]).

In Fig. 1, we show the top-level of our pattern matching architecture on FPGA. Our architecture consists of an input decoder, a collection of pattern matching modules, and an output encoder. It receives and sends a sequence of I/O packets from and to a host PC through a fast bus such as PCI Express. In the present implementation, I/O packets have 64-bit length.

Our architecture runs with two different modes: the *pre-processing mode* and the *run-time mode*. In the preprocessing mode, our architecture loads the description of input patterns with packets and in the run-time mode, receives an input letter, makes a state transition for the target NFA by a fixed circuitry of a pattern matching module, detects matches, and emits a match information by receiving and sending packets.

IV. Pattern matching module

A *pattern matching module* (PMM, for short) is a core of our architecture and efficiently simulates the NFA of an input pattern. The construction of a PMM depends on the class of an input pattern. In what follows, we give the construction of a PMM for the class EXT of extended patterns with fixed length L according to a bit-parallel method for EXT, called the Extended SHIFT-AND method [9]. In Fig. 2, we show the circuit of a pattern matching module for EXT.

Expanded form and bit-assignment. Let R be an extended pattern. Then, recall that every component r_i of R has one of the following types: (i) $r_i = \alpha$, (ii) $r_i = \alpha?$, (iii) $r_i = \alpha^*$, (iv) $r_i = \alpha^+$, and (v) $r_i = \alpha\{x, y\}$, where $\alpha \subseteq \Sigma$. Note that a letter $a \in \Sigma$ and a wildcard '.' are classes of letters of type (i). First, we expand all occurrences of bounded repeats $\alpha\{x, y\}$ of type (v) in R by using the equivalence $\alpha\{x, y\} \equiv (\alpha?)^{y-x}\alpha^x$, where $x \leq y$. Let $\textsc{Expand}(R) = r_1 \cdots r_m$ be the resulting expanded form of R consisting of m components, where $|R| \leq m \leq L$. By the construction, the expanded form $\textsc{Expand}(R)$ contains no occurrences of components of type (v). Then, we assign the unique numbers $\mathcal{I} = \{1, \cdots, m\}$, called the *bit-positions*, to all components of $\textsc{Expand}(R) = r_1, \ldots, r_m$.

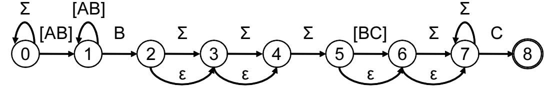

Fig. 3. The extended pattern NFA $N(R_2)$ of $R_2 = \mathtt{[AB]^+B.\{1,3\}[BC]?.^*C}$.

For example, let $R_2 = \mathtt{[AB]^+B.\{1,3\}[BC]?.^*C}$ be the target extended pattern consisting of six components. Then, by replacing the bounded gap $.\{1, 3\}$ with $(.?)(.?)(.)$, we obtain its expanded form $\textsc{Expand}(R_2) = (\mathtt{[AB]^+})(\mathtt{B})(.?)(.?)(.)(\mathtt{[BC]?})(.^*)(\mathtt{C})$ consisting of eight components with assigned bit-positions from 1 to 8.

Construction of NFA. Then, we obtain the *extended pattern NFA* $N_R = N(R)$ for R from the expanded form $\textsc{Expand}(R)$ as follows. Let $\textsc{Expand}(R) = r_1 \cdots r_m$ for some $m \geq 1$ and L be an positive integer larger than or equal to than m. L is actually the bit-length of registers in an underlying hardware. By construction, we can assume that $\textsc{Expand}(R)$ contains components of only type (i)–(iv). For every $i = 1, \ldots, m$, we add to the NFA N_R edges related to state i according to the type of the i-th component r_i with matrix α as follows:

- For all types (i)–(iv) of r_i, we add the backbone edge $e_i = (i-1, \alpha, i)$ directed from the previous state $i-1$ to the current state i labeled with matrix α.
- Furthermore, if r_i is either (ii) $\alpha?$ or (iii) α^*, then we add an ε-edge directed from the previous state $i-1$ to the current state i.
- Furthermore, if r_i is either (iii) α^* or (iv) α^+, then we add a self-loop labeled with matrix α from the current state i to itself.

For the expanded form $\textsc{Expand}(R) = r_1 \cdots r_m$, an *$\varepsilon$-block* in $\textsc{Expand}(R)$ is the set $B = \{i, i+1, \ldots, j\} \subseteq \mathcal{I}$ of the component indexes for a maximal consecutive subsequence $r_{i+1} \cdots r_j$ $(1 \leq i \leq j \leq m)$ connected with ε-edges, where r_k is either $r_k = \alpha_k?$ or $r_k = \alpha_k^*$ for some $i \leq k \leq j$. Let $B_1, \ldots, B_h$ $(h \geq 0)$ be the ε-blocks of $\textsc{Expand}(R)$. For

Bit-positions	1	2	3	4	5	6	7	8
R	[AB]$^+$	B	.{1,3}			[BC]?	.*	C
EXPAND(R)	[AB]$^+$	B	.?	.?	.	[BC]?	.*	C

Fig. 4. The bit-position assignment for R_2 = [AB]$^+$B.{1,3}[BC]?.*C and its expanded form EXPAND(R_2).

Bit-position i	1	2	3	4	5	6	7	8
INIT	1	0	0	0	0	0	0	0
ACCEPT	0	0	0	0	0	0	0	1
MOVE[A]	1	0	1	1	1	0	1	0
MOVE[B]	1	1	1	1	1	1	1	0
MOVE[C]	0	0	1	1	1	1	1	1
MOVE[%]	0	0	1	1	1	0	1	0
REPPOS[A]	1	0	0	0	0	0	1	0
REPPOS[B]	1	0	0	0	0	0	1	0
REPPOS[C]	0	0	0	0	0	0	1	0
REPPOS[%]	0	0	0	0	0	0	1	0
EpsBEG	0	1	0	0	1	0	0	0
EpsEND	0	0	0	1	0	0	1	0
EpsBLK	0	1	1	1	1	1	1	0

Fig. 5. The set of bit-masks for R_2 = [AB]$^+$B.{1,3}[BC]?.*C on alphabet $\Sigma = \{$A, B, C$\}$, where the symbol '%' denotes any letter not in Σ.

example, we show in Fig. 3 the extended pattern NFA $N(R_2)$ corresponding to EXPAND(R_2). Then, EXPAND(R_2) has two ε-blocks $B_1 = \{2, 3, 4\}$ and $B_2 = \{5, 6, 7\}$ corresponding to $r_3r_4 = (.?)(.?)$ and r_6r_7 = ([BC]?)(.*), respectively.

Construction of bit-masks. To simulate an extended pattern NFA N_R, we encode its information in the following five L-bit masks and two arrays of L-bit masks stored in a collection of registers and block RAMs, respectively:

- INIT is the L-bit mask that sets 1 at the bit-position for state 1. That is, INIT$[i] = 1$ if and only if $i = 1$.
- ACCEPT is the L-bit mask that sets 1 at the bit-position for the final state m. That is, ACCEPT$[i] = 1$ if and only if $i = m$.
- EpsBEG is the L-bit mask that sets 1 at the lowest bit-position of every ε-block. That is, EpsBEG$[i] = 1$ if and only if $i = \min(B_k)$ for some ε-block B_k.
- EpsEND is the L-bit mask that sets 1 at the highest bit-position of every ε-block. That is, EpsEND$[i] = 1$ if and only if $i = \max(B_k)$ for some ε-block B_k.
- EpsBLK is the L-bit mask that sets 1s at all bit-positions in every ε-block. That is, EpsBLK$[i] = 1$ if and only if i is contained by some ε-block B_k.
- MOVE$[a]$ is the L-bit mask that indicates all bit-positions of backbones labeled with a letter $a \in \Sigma$ in EXPAND(R). That is, MOVE$[a][i] = 1$ if and only if the state i has a backbone labeled with $a \in \alpha$, where α is the matrix of component r_i.
- REPPOS$[a]$ is the L-bit mask that indicates all bit-positions of self-loops labeled with a letter $a \in \Sigma$ in EXPAND(R). That is, REPPOS$[a][i] = 1$ if and only if the state i has a self-loop labeled with $a\alpha$, or equivalently, either $r_i = \alpha^*$ or $r_i = \alpha^+$ with $a \in \alpha$.

Cycle i	Input letter t_i	STATE after update in cycle i: 1	2	3	4	5	6	7	8	Emit Match
1	A	1	0	0	0	0	0	0	0	0
2	B	1	1	1	1	0	0	0	0	0
3	C	0	0	1	1	1	1	1	0	0
4	B	1	0	0	1	1	1	1	0	0
5	B	1	1	1	1	1	1	1	0	0
6	C	0	0	1	1	1	1	1	1	1

Fig. 6. An example of extended pattern matching, given an extended pattern R_2 = [AB]$^+$B.{1,3}[BC]?.*C and an input text T = ABCBBC.

In Fig. 4 and Fig. 5, we show examples of the bit-position assignment and the bit-masks for R_2 = [AB]$^+$B.{1,3}[BC]?.*C, respectively. We store the bit-masks INIT, ACCEPT, EpsBEG, EpsEND, and EpsBLK in L-bit registers, and the arrays $(\texttt{MOVE}[a])_{a\in\Sigma}$ and $(\texttt{REPPOS}[a])_{a\in\Sigma}$ in block RAMs.

Control logic for NFA-simulation. We finally give the control logic for simulation of an extended pattern NFA N_R. Fig. 2 shows the whole circuit of NFA-simulation for EXT. This circuit is based on the codes of the Extended SHIFT-AND method [9].

First, the following code initializes the bit-mask STATE that represents a set of active states at line (1) and applies the letter transitions by backbones and self-loops labeled with t at lines (2) and (3), respectively, where $t \in \Sigma$ is the current input letter in an input text.

$$\texttt{STATE} \leftarrow (((\texttt{STATE} \ll 1) \mid \texttt{INIT}) \quad (1)$$
$$\& \ \texttt{MOVE}[t]) \quad (2)$$
$$\mid (\texttt{STATE} \ \& \ \texttt{REPPOS}[t]); \quad (3)$$

Then, we simulate ε-transitions by the next codes. At line (4), we turn on the highest bit of each ε-block in STATE, and set it to HIGH. At line (5), for each ε-block in HIGH, we invert all bits lower than or equal to the lowest 1 bit of each ε-block in HIGH and set it to LOW. At line (6), the resulting bit-mask has 1s at all bit-positions properly higher than the lowest 1 of each ε-block in STATE. Finally, we add the change to STATE at line (7).

$$\texttt{HIGH} \leftarrow \texttt{STATE} \mid \texttt{EpsEND}; \quad (4)$$
$$\texttt{LOW} \leftarrow \texttt{HIGH} - \texttt{EpsBEG}; \quad (5)$$
$$\texttt{STATE} \leftarrow (\texttt{EpsBLK} \ \& \ ((\sim \texttt{LOW}) \oplus \texttt{HIGH})) \quad (6)$$
$$\mid \texttt{STATE}; \quad (7)$$

The acceptance test is given by the following code.

$$\textbf{if } (\texttt{STATE} \ \& \ \texttt{ACCEPT}) \textbf{ then } \texttt{EmitMatch} \leftarrow 1;$$

In Fig. 6, we show an example of NFA-simulation by the set of bit-masks for R_2 = [AB]$^+$B.{1,3}[BC]?.*C on an input text T = ABCBBC. In the figure, we show the status of the bit-mask STATE after the update in each cycle i ($1 \le i \le 6$). The output EmitMatch of a PMM is the value at the bit-position 8 of STATE.

TABLE I
Summary of parameters of the proposed pattern matching hardwares. See the text for the meanings of the parameter names.

Class of Patterns	#Op	#Add	#Reg	#BL	#Slice	Freq	Throughput	Load Time Total	#Patterns	#Chars Total
Exact string patterns	5	0	3	256	54	363 MHz	2.9 Gbps	0.182 msec	256	8,192 letters
Extended patterns	12	1	6	512	123	202 MHz	1.6 Gbps	0.328 msec	128	4,096 letters

TABLE II
Results comparisons of regular expression matching hardwares based on various dynamically reconfigurable architectures, where Class indicates the target class, bRAM/char the number of bytes used in block RAMs per letter, and LC/char the number of logic cells used per letter.

Design	Class	Device	Throughput	bRAM/char	LC/char	#Chars Total
Dynamic BP-NFA for STR [7]	STR	Virtex-5 LX330	2.9 Gbps	4 bytes/char	10.8 LC/char	8192
Dynamic BP-NFA for EXT [7]	EXT	Virtex-5 LX330	1.6 Gbps	8 bytes/char	24.6 LC/char	4096
KMP-based hardware [4]	STR	Virtex-II Pro	1.8 Gbps	4 bytes/char	3.2 LC/char	3200
Bitsplit-based hardware [5]	STR	Virtex-4 FX100	1.6 Gbps	46 bytes/char	1.4 LC/char	16715
RegExp Controller hardware [3]	REG	Virtex-4 FX100	1.4 Gbps	46 bytes/char	2.56 LC/char	16715

V. Experimental Results

To evaluate the time and area complexities, we implemented our pattern matching hardwares Dynamic BP-NFA in Verilog HDL for both classes STR and EXT of strings and extended patterns, where we set $L = 32$ and the arrays MOVE and REPPOS are implemented in block RAMs. Note that a PMM for STR has INIT, ACCEPT, and $(\mathtt{MOVE}[a])_{a\in\Sigma}$. For the details of a PMM for STR, see [7]. We targeted the Virtex-5 LX330 with -1 speed grade, which has 51,840 slices and 288 block RAMs with 36 Kbits. We used the Xilinx ISE Design Suite 10.1 and Synopsys VCS development tools. For comparison, we also implemented a hardware in the static compilation approach, called Static BP-NFA [8], based on the SHIFT-AND method [2] for STR on Xilinx Virtex-5 LX50 with 7,200 slices. For the details, see [8].

A. Results on our dynamically reconfigurable hardwares

In Table. I, we show the summary of the experimental results of a single pattern matching module, PMM, in our dynamically reconfigurable hardwares Dynamic BP-NFA, where #Op, #Add, #Reg, #BL, and #Slice are the numbers of 32-bit operations, 32-bit integer additions, registers, block RAM lines per PMM, respectively. The number of block RAM lines (#BL) is given by the number of block RAMs times $|\Sigma| = 256$. #Patterns and #Chars Total are the number and the total size of input patterns, respectively. Below, we give results on our Dynamic BP-NFA and its comparisons against our previous static compilation hardware Static BP-NFA [8].

Performance evaluation. The maximum frequencies of our hardwares were 363 MHz and 202 MHz, respectively, for STR and EXT. For the time complexity in the run-time, we estimated the throughput of matching by Throughput $=$ Freq$\times 8$ (bit/sec). Thus, the throughputs were 2.9 Gbps and 1.6 Gbps since our hardwares consume one letter (8 bits) per clock. On the other hand, the Static BP-NFA hardware achieved frequency of 216 MHz and throughput of 1.7 Gbps for 300 patterns in STR using 2,925 slices. Therefore, the dynamic version is faster than the static version.

Resource usage. For the Dynamic BP-NFA, we could implement up to 256 PMMs ($8,192$ total letters) and up to 128 PMMs ($4,096$ total letters), for STR and EXT, respectively. In this setting, the place-and-route took around one hour in both classes. For EXT, one PMM used 123 slices (total 15,744 slices) and two block RAMs ($512 = 2\times256$ lines were used). Consequently, the usage of block RAMs was 89%, while the usage of slices was only 24%. This means that the size of a hardware in our architecture is constrained mainly by the amount of block RAMs and not by one of the slices. On the other hand, the Static BP-NFA used much less resources than the Dynamic BP-NFA. We could implement up to 1,500 PMMs (around 20K total letters) for STR using $7,200$ slices (100%) and no block RAMs, where the slice usage seems linear in the number N of PMMs for $N = 1$ to 500 and seems almost constant for $N = 500$ to $1,500$ ([8]).

Reconfiguration time comparison. We estimated the total loading time of a dynamically reconfigurable architecture by Load Time Total $=$ #Patterns$\times$(#Reg$+$#BL)$/$Freq (sec). From the result of Table I, the Dynamic BP-NFA took 0.71 μsec and 2.56 μsec per PMM to load an input pattern, and consequently, took the total loading times 0.182 msec and 0.328 msec to load all 256 and 128 patterns for STR and EXT, respectively. On the other hand, we estimated the loading time of the Static BP-NFA by the compilation time including place-and-route. By experiments, it required 4.27×10^5 msec for 300 PMMs, approximately seven minutes. Hence, the Dynamic BP-NFA is 10^6 times faster than the Static BP-NFA in load time.

B. Comparison against other pattern matching hardwares

In Table. II, we compared our hardwares against the previous dynamically reconfigurable hardwares [3], [4], [5].

Performance evaluation. For the class STR, our hardware achieved higher throughput of 2.9 Gbps than Baker *et al.*'s KMP-based hardware [4] and Jung *et al.*'s Bitsplit-based hardware [5]. For more general classes of regular expressions, Baker *et al.*'s RegExp Controller hardware [3], which is a hybrid of DFA-simulation and microcontroller, has been the only dynamically reconfigurable architecture for a non-trivial

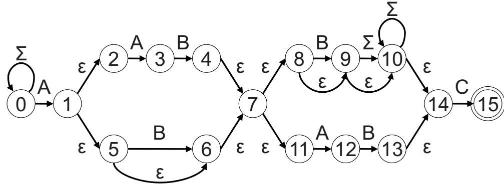

Fig. 7. The Thompson NFA $N(R_4)$ of an extended network expression $R_4 = \mathtt{A(AB|B?)(B?.*|AB)C}$.

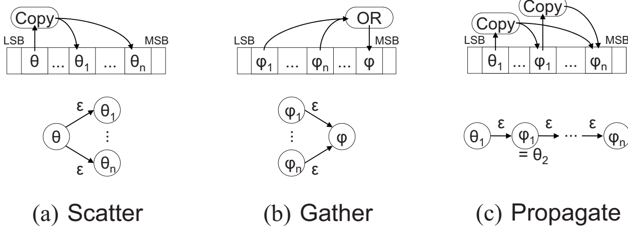

Fig. 8. The bit-operations and the corresponding parts of TNFAs.

subclass of REG so far. Compared with the hardware by [3], our hardware achieved comparable throughput of 1.6 Gbps for EXT to 1.4 Gbps of [3]. From the above results, our hardware and the hardware by [3] were comparable in the term of running time. An advantage of our hardware is that it has the worst-case guarantee of its throughput regardless of an input stream, while the hardware by [3] can overflow when many matches of short segments occur.

We also implemented our PMMs for STR and EXT on the same device, Virtex-4 FX100 with -12 speed grade, as [5] and [3]. Then, our hardwares for STR and EXT also achieved higher throughput of 3.6 Gbps than the hardware by [5] and comparable throughput of 1.6 Gbps to the hardware by [3], respectively.

Resource usage. Our hardwares load only a set of bit-masks that represents the information of regular expressions into registers and block RAMs, while both of Jung *et al.*'s Bitsplit-based hardware [5] and Baker *et al.*'s RegExp Controller hardware load a state transition table and a sequence of instructions into block RAMs, respectively. Therefore, our hardwares required less block RAM lines than the hardwares by [5] and [3]. On the other hand, our hardwares required ten times more logic cells (slices) than [3] since our hardwares required many ALUs to implement the circuitry for NFA-simulation as shown in Fig. 2.

VI. Extension to Network and Extended Network Expressions

In Section IV, we gave the construction of a pattern matching module (PMM) for the class EXT of extended patterns based on the Extended SHIFT-AND method [9]. To extend the PMM for EXT to more complex classes, we developed a fast bit-parallel method, called the *Extended^2 SHIFT-AND method*, for the classes NET and EXNET of network and extended network expressions [6].

For this method, we devise new bit-parallel operations, called *Scatter*, *Gather*, and *Propagate*, for efficient simulation of a *Thompson NFA* (TNFA, for short) [9] for NET and EXNET, which can have the long succession and the branching of ε-edges caused by concatenation and union as shown in Fig. 7. In Fig. 8, we show each bit-operation and the corresponding part of a TNFA that the operation simulates. Furthermore, we also devise a transformation technique of a TNFA, called *bypassing*, into a special form of NFA that satisfies a property, called *bimonotonicity*, of an ε-path by attaching new ε-edges to all subexpressions whose initial and final states are ε-reachable in the original expression.

We implemented a PMM for EXNET based on our *Extended^2 SHIFT-AND method*. The experimental results showed that our hardware for EXNET achieved the throughput of 0.5 Gbps.

VII. Conclusion

In this paper, we report recent progress on dynamically reconfigurable hardwares on FPGA for high-speed string and regular expression matching. For complex subclasses of regular expressions such as extended strings, network expressions, and extended network expressions, our architecture based on bit-parallel NFA-simulation method provides fast dynamic reconfiguration of the patterns as well as high-throughput pattern matching.

In experiment, we showed that our dynamically reconfigurable hardwares achieved throughputs of 2.9 Gbps and 1.6 Gbps for the classes of STR and EXT, respectively. In the present DPI (deep packet inspection) technology, where only the first several kilobytes of each packet is scanned, our hardwares' throughputs look enough. However, further progress of DPI technology will require higher throughput. Therefore, it is a future research to make further speed-up of our hardwares.

In this paper, we considered end match information only. In recent applications such as ESP, however, it is also important to consider interval matches and non-overlapping matches. Therefore, it is an interesting problem to extend our hardwares to report matching information of both types.

Acknowledgment

The authors would like to thank Satoshi Kamiya in NEC Corp., Yoshikazu Miyanaga in Hokkaido Univ., Shinobu Nagayama in Hiroshima City Univ., and Osamu Watanabe in Tokyo Inst. Tech. for their discussions and valuable comments. This research was partly supported by MEXT Grant-in-Aid for Scientific Research (A), 20240014, FY2008–2011, "*High performance FPGA-based string matching hardwares*" project under MEXT/JSPS Global COE Program at IST, Hokkaido University, FY2007–2011, and ERATO MINATO Discrete Structure Manipulation System Project, JST.

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, “Efficient pattern matching over event streams,” in *Proc. SIGMOD’08*, pp. 147–160, 2008.

[2] R. Baeza-Yates and G. H. Gonnet, “A new approach to text searching,” *CACM*, 35(10), pp. 74–82, 1992.

[3] Z. K. Baker, H. Jung, and V. K. Prasanna, “Regular expression software deceleration for intrusion detection systems,” in *Proc. FPL’06*, pp. 1–8, 2006.

[4] Z. K. Baker and V. K. Prasanna, “Time and area efficient pattern matching on FPGAs,” in *Proc. ACM FPGA’04*, pp. 223–232, 2004.

[5] H. J. Jung, Z. K. Baker and V. K. Prasanna, “Performance of FPGA implementation of bit-split architecture for intrusion detection systems,” in *Proc. RAW’06*, 2006.

[6] Y. Kaneta, S. Minato, and H. Arimura, “Fast bit-parallel matching for network and regular expressions,” in *Proc. SPIRE’10*, pp. 372–384, 2010.

[7] Y. Kaneta, S. Yoshizawa, S. Minato, H. Arimura, and Y. Miyanaga, “Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching,” in *Proc. FPT’10*, pp. 21–28, 2010.

[8] Y. Kaneta, S. Yoshizawa, S. Minato, H. Arimura, and Y. Miyanaga, “Efficient multiple regular expression matching on FPGAs based on extended SHIFT-AND method,” in *Proc. SASIMI’10*, pp. 401–406, 2010.

[9] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*, Cambridge University Press, 2002.

[10] H. Roan, W. Hwang, and C. D. Lo, “Shift-or circuit for efficient network intrusion detection pattern matching,” in *Proc. FPL’06*, pp. 1–6, 2006.

[11] R. Sidhu and V. K. Prasanna, “Fast regular expression matching using FPGAs,” in *Proc. IEEE FCCM’01*, pp. 227–238, 2001.

[12] S. Wu and U. Manber, “Fast text searching: allowing errors,” *CACM*, 35(10), pp. 83–91, 1992.

[13] Y. E. Yang and V. K. Prasanna, “Memory-efficient pipelined architecture for large-scale string matching, ” in *Proc. IEEE FCCM’09*, pp. 104–111, 2009.

[14] Y. E. Yang, W. Jiang, and V. K. Prasanna. “Compact architecture for high-throughput regular expression matching, ” in *Proc. ACM/IEEE ANCS’09*, pp. 30–39, 2009.