PAPER

A Dynamically Reconfigurable FPGA-Based Pattern Matching Hardware for Subclasses of Regular Expressions

Yusaku KANETA[†*a)], ***Student Member***, **Shingo YOSHIZAWA**[††b)], **Shin-ichi MINATO**[†,†††c)], **Hiroki ARIMURA**[†d)], ***Members***, ***and*** **Yoshikazu MIYANAGA**[†e)], ***Fellow***

SUMMARY In this paper, we propose a novel architecture for large-scale regular expression matching, called *dynamically reconfigurable bit-parallel NFA architecture* (Dynamic BP-NFA), which allows dynamic loading of regular expressions on-the-fly as well as efficient pattern matching for fast data streams. This is the first dynamically reconfigurable hardware with guaranteed performance for the class of extended patterns, which is a subclass of regular expressions consisting of union of characters and its repeat. This class allows operators such as character classes, gaps, optional characters, and bounded and unbounded repeats of character classes. The key to our architecture is the use of *bit-parallel pattern matching* approach, in which the information of an input non-deterministic finite automaton (NFA) is first 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 bitwise Boolean and arithmetic operations consuming one input character per clock regardless of the actual contents of an input text. Experimental results showed that our hardwares for both string and extended patterns were comparable to previous dynamically reconfigurable hardwares in their performances.

key words: *FPGA, string matching, regular expression matching, bit-parallel algorithm, event stream processing*

1. Introduction

1.1 Background

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

Manuscript received November 29, 2011.
Manuscript revised March 12, 2012.

[†]The authors are with the Graduate School of Information Science and Technology, Hokkaido University, Sapporo-shi, 060–0814 Japan.

[††]The author is with the Department of Electrical and Electronic Engineering, Kitami Institute of Technology, Kitami-shi, 090–8507 Japan.

[†††]The author is with ERATO MINATO Discrete Structure Manipulation System Project, Japan Science and Technology Agency, at Hokkaido University, Sapporo-shi, 060–0814 Japan.

[*]Presently, with Rakuten Institute of Technology, Rakuten, Inc.

a) E-mail: y-kaneta@ist.hokudai.ac.jp
b) E-mail: yosizawa@mail.kitami-it.ac.jp
c) E-mail: minato@ist.hokudai.ac.jp
d) E-mail: arim@ist.hokudai.ac.jp
e) E-mail: miya@ist.hokudai.ac.jp

DOI: 10.1587/transinf.E95.D.1847

The *large-scale pattern matching problem* [3], [5], [13], [15], [20], [24], [26] 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., from thousands to several tens thousands) of *complex patterns* (e.g., regular expressions) against *high-speed* data streams (e.g., of several giga bps). For example, large-scale regular expression matching is such a problem that appears in real-world applications such as ESP and NIDS, where regular expressions are defined by characters, concatenation, union, and Kleene-star. From the view of current CPU technologies, large-scale pattern matching problems are quite CPU-intensive tasks, and thus it is difficult for software on CPU to efficiently process massive data streams in real time. Therefore, research on large-scale regular expression matching on reconfigurable hardwares such as *field programmable gate arrays* (*FPGAs*) have recently attracted much attention.

1.2 Dynamic Reconfiguration vs. Static Compilation Approaches

A recent research trend in large-scale regular expression matching hardwares is to simulate finite state automata for a class of regular expressions on a specially designed hardware [3], [4], [8], [13], [19], [20], [24]–[26]. This approach is further classified into the *static compilation approach* and the *dynamic reconfiguration approach*.

In the static compilation approach [19], [20], [24]–[26], a set of input regular expressions are transformed into either deterministic finite automata (DFAs) or non-deterministic finite automata (NFAs) [16], and then statically compiled into wired logic on FPGA. However, the static compilation approach has a drawback in that modification of regular expressions is too expensive to be done frequently.

In the dynamic reconfiguration approach [3], [4], [8], [13], a universal control logic is statically compiled into FPGA beforehand as well, but a description of regular expressions is dynamically loaded to the FPGA as data in preprocessing, and then simulated in run-time. This approach is attractive in real-world applications such as ESP and NIDS, where input patterns frequently change. However, it is a challenging task to design dynamically reconfigurable hardwares that efficiently run for wider classes of regular expressions since the classes of patterns that can be dealt with are still limited.

Overall, our research goal is to design dynamically re-

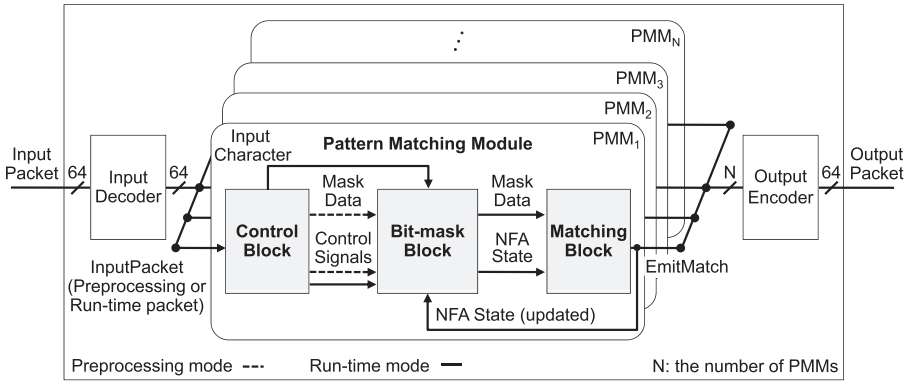

Fig. 1 The top-level view of our pattern matching architecture.

configurable hardwares for complex classes of regular expressions and to achieve gigabit throughput as the hardwares by [3], [4], [6], [8], [14], [21].

1.3 Main Result of This Paper

As a main result, we propose a novel architecture for large-scale regular expression matching, called *dynamically reconfigurable bit-parallel NFA architecture* (Dynamic BP-NFA), where quick on-the-fly loading of complex pattern is possible. The key to our architecture is the use of *bit-parallel pattern matching* approach developed in string matching communities since 1990s [2], [16], [23]. In Fig. 1, we show the top-level view of our architecture, which will be explained in detail.

Our target pattern class is the class EXT of *extended patterns* [16], [17], which is a subclass of regular expressions allowing union of characters and its Kleene-star. The class EXT is a natural generalization of the class STR of *exact string patterns* consisting characters and concatenation, which supports character classes $\beta = [ab\cdots]$, gaps '.', optional characters β?, bounded and unbounded repeats $\beta\{x, y\}$ and β^* of character classes. For example, $R = \texttt{[AB]}^+\texttt{B.\{1, 3\}[AC]?.}^*\texttt{C}$ represents a substring of an input text that starts with one or more repetitions of character A or B, followed by a character B, one to three repetitions of any character in Σ, an optional A or C, any characters in Σ, and C.

Our Dynamic BP-NFA consists of a collection of pattern matching modules. In each module, an input extended pattern is first translated into an input NFA, and the information of the NFA is compactly encoded in a set of bit-masks stored in 32-bit registers and block RAMs, when the underlying register length is 32 bits. Then, the NFA is efficiently simulated by a fixed circuitry using a set of 32-bit Boolean operations and a 32-bit integer addition on the registers and RAMs. We apply the same procedure to a set of input patterns in multiple pattern matching. Based on theoretical analysis, we show that this hardware correctly matches a given set of extended patterns against an input text consuming one input character per clock regardless of the actrual contents of the input texts.

In experiments, we first implemented our Dynamic BP-NFA and measured its performance and resourse usage for STR and EXT. As results, for STR, we could install 256 patterns and achieve the throughput of 2.6 Gbps on Xilinx Virtex-5 LX330, and for EXT, we could install 128 patterns and achieve the throughput of 1.4 Gbps on the same device. Next, we compared the performances of our hardwares for both STR and EXT to those of the previous dynamically reconfigurable hardwares in the literatures [3], [4], [8] after a calibration of throughputs using process scaling in CMOS technologies that FPGA devices were built on. Consequently, our hardwares for both classes were comparable to those of the above dynamically reconfigurable hardwares in their performances.

Main contributions of this paper are summarized as follows:

- *Dynamic loading of extended patterns*: Our Dynamic BP-NFA is the first dynamically reconfigurable hardware that can deal with the class EXT of extended patterns, which is a non-trivial and useful subclass of regular expressions widely studied in real-world applications such as event stream processing [1] and bioinformation processing [16], [17]. On the other hand, Baker *et al.*'s KMP-based hardware [4] and Jung *et al.*'s Bitsplit-based hardware [8] are also dynamically reconfigurable. However, they allow only dynamic loading of the class STR of exact string patterns, which is a small subclass of EXT. While Sidhu *et al.*'s hardware [20] can deal with the whole class REG of regular expressions, it can not change input regular expressions on-the-fly since it is a static compilation approach.
- *Worst-case performance guarantee*: Our Dynamic BP-NFA has theoretical performance guarantee in the worst-case from Theorem 1 in Sect. 4.3 for both STR and EXT. On the other hand, Baker *et al.*'s RegExp Controller hardware [3] is only a dynamically reconfigurable hardware before ours that can deal with a non-trivial subclass of regular expressions. Since it is a hybrid of DFA-simulation and microcontroller, its processing time related to microcontroller becomes a performance bottleneck and it has no performance guarantee in the worst-case when a regular expression has

many occurrences of its subpatterns in an input text as indicated in [3].

- *Potential extensibility of target pattern classes*: Our Dynamic BP-NFA has the potential extensibility to more general patterns by changing the construction of bit-masks and the control logic for NFA-simulation. For example, Kaneta *et al.* [10] proposed a generalization of the Extended SHIFT-AND method, used in this paper, for the classes of network and regular expressions allowing union and Kleene-star. Such method can be incorporated into our architecture by extending the construction of bit-masks and a circuitry described later.

Overall, our Dynamic BP-NFA is the first architecture that fulfills the above three requirements for dynamic loading of patterns, worst-case performance guarantee, and extensibility to more general patterns.

1.4 An Example of Large-Scale Pattern Matching

Deep packet inspection (*DPI*) [6], [14], [21] is a new technology in NIDS that scans the payloads deep inside packets using a set of regular expressions as detection rules for detecting network intrusion, while traditional NIDS makes shallow analysis only in the headers of input packets. DPI requires efficient solutions for large-scale pattern matching problems. Among present DPI systems in NIDS, Snort system [22] is one of the most widely used deep packet inspection systems. The current version of Snort system, at the time of May 2011, handles more than twenty thousands of detection rules in *Perl-Compatible Regular Expressions* (*PCRE*) [18].

Although the present Snort system is a software-based DPI system running on CPU and its performance seems adequate for the current applications, it is recognized that further progress of high-speed network technology will require order of magnitude faster hardware implementation [3], [6], [14], [21]. Such hardwares are desired to have the capability of on-the-fly reconfiguration of patterns as well as high-performance guarantee for a wide class of regular expressions. However, most of the currently available dynamically reconfigurable hardwares [4], [8] can deal with only exact string patterns. Interestingly enough, we can observe that most of detection rules are either extended patterns in EXT, which will be considered in this paper, or disjunctions of a few extended patterns. Hence, from the view point in DPI, our architecture will be a candidate of the base technologies for such high-performance hardware-based DPI systems.

1.5 Related Work

There have been a number of research on dynamically reconfigurable hardwares for large-scale pattern matching. Baker *et al.*'s KMP-based hardware [4] and Jung *et al.*'s Bitsplit-based hardware [8] are DFA-based dynamic reconfiguration approaches for exact string patterns, which achieved the throughputs of 1.8 Gbps and 1.6 Gbps, respectively. Baker *et al.* developed RegExp Controller hardware for regular expressions [3], which is a hybrid of DFA-simulation and microcontroller. The throughput of this hardware was 1.4 Gbps.

There is also a line of research [14], [21] that study heuristics for converting a given NFA into a compact DFA without the state explosion of DFAs. Kumar *et al.* [14] developed a regular expression matching hardware based on *delayed input deterministic finite automata* (D^2 FAs) and Smith *et al.* [21] presented a chip design based on *extended finite automata* (XFAs), where both of them achieved the high throughputs by parallel computation. Dharmapurikar *et al.* [6] studied a hardware-based Bloom filter, which achieved 2.5 Gbps for exact string patterns with the assumption that a match occurs with a small probability.

1.6 Organization of This Paper

This paper is organized as follows. In Sect. 2, we give basic definitions. In Sect. 3, we propose our Dynamic BP-NFA architecture, and in Sect. 4, we give the detailed description of each pattern matching module. In Sect. 5, we give experimental results, and in Sect. 6, we conclude. This paper was built on the previous publications [11], [12].

2. Preliminary

2.1 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 *characters* (or *letters*). A *string* on Σ is a sequence $S = s_1 \cdots s_n$ of characters, 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 $S[i..j] = \varepsilon$. We denote by Σ^* the set of all strings on Σ. 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. For a character $c \in \Sigma$ and an integer $i \in \mathbf{N}$, we define by c^i the string consisting of i consecutive c.

Let REG be the class of regular expressions on Σ. More precisely, a regular expression R is either a character $c \in \Sigma$, concatenation $R = R_1 \cdot R_2$, union $R = (R_1|R_2)$, and Kleene-star $R = (R_1)^*$, where R_1 and R_2 are regular expressions [16]. For a regular expression R, 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 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 $C \subseteq$ REG of regular expressions is defined as follows. An input is an input *pattern set* $\mathcal{P} = \{ (i, R_i) \mid i = 1, \ldots, N \}$ $(N \geq 1)$, where for every $i = 1, \ldots, N$, i is an in-

[†]In the case that $n < 1$, a text $T = t_1 \cdots t_n$ represents the empty string ε. Similarly, a set $S = \{s_1, \ldots, s_m\}$ and a regular expression $R = r_1 \cdots r_m$ represent the empty set $\emptyset$ and the empty string ε if $m < 1$, respectively.

teger, called an *index*, and $R_i \in C$ is a pattern. Then, the task is, given a stream $T = t_1 t_2 \cdots t_p \cdots$ $(p \geq 1)$ of input characters, to output the pairs (i, p) such that $i = 1, \ldots, N$ is the index and p is an end position of R_i in T for all $p = 1, 2, \ldots$.

2.2 Target Pattern Class: Extended Patterns

The target subclass of regular expressions that our architecture deals with is the class of *extended patterns* defined as follows. In what follows, we directly write a set $\{a_1, \ldots, a_m\} \subseteq \Sigma$ of characters instead of union $(a_1|\cdots|a_m)$ of characters representing the set, where $a_i \in \Sigma$ $(1 \leq i \leq m)$.

Definition 2. The class of *extended patterns*, denoted by EXT, is a subclass of regular expressions defined as follows: an extended pattern R on Σ is a sequence of some components $R = r_1 \cdots r_m$ $(m \geq 0)$, where for each $1 \leq i \leq m$, r_i is an expression, called a *component*, with one of the following forms, where $\equiv$ means the notational equivalence:

(1) A *character* (or a *letter*) $r_i = c \in \Sigma$ is a component with the language $L(c) = \{c\}$.
(2) A *gap* (or a *don't care*) $r_i = .$ is a component with the language $L(.) = \Sigma$. This matches any character in Σ.
(3) A *character class* $r_i = \beta$ is a component with the language $L(\beta) = \beta$, where $\beta \subseteq \Sigma$. As notation, we write $[ab\cdots]$ for $\beta = \{a, b, \ldots\}$. This is equivalent to union of characters. Note that a character $a \in \Sigma$ and a gap '.' are character classes.
(4) An *optional character* $r_i = \beta?$ is a component, where $\beta \subseteq \Sigma$ and $\beta? \equiv (\beta|\varepsilon)$.
(5) *Bounded repeats* $r_i = \beta\{x, y\}$, $r_i = \beta\{, y\}$, and $r_i = \beta\{x\}$ are components with equivalence $\beta\{x, y\} \equiv (\beta?)^{y-x}\beta^x$, $\beta\{, y\} \equiv (\beta?)^y$, and $\beta\{x\} \equiv \beta^x$, respectively, where $\beta \subseteq \Sigma$ and $x \leq y$ $(x, y \in \mathbf{N})$. If β is a gap '.', r_i is called a *bounded gap*.
(6) *Unbounded repeats* $r_i = \beta^*$ and $r_i = \beta^+$ are components, where $\beta \subseteq \Sigma$ and $\beta^+ \equiv \beta\beta^*$. If β is a gap '.', r_i is called an *unbounded gap* (or a *variable length don't care*).

For $R = r_1 \cdots r_m$, we define its language by $L(R) = L(r_1) \cdots L(r_m)$. If r_i is one of the forms $\beta?$, $\beta\{x, y\}$, β^*, and β^+, then β is called the *matrix* of r_i.

Example 1. *We show examples of extended patterns, where character class is equivalent to union of characters, e.g.,* `[AB]` ≡ `(A|B)`.

- R_1 = `ABABBC`.
- R_2 = `[AB]`$^+$`B.{1, 3}[AC]?.`*`C`.
- R_3 = `(A[BC]`*`).{, 4}([DE]`$^+$`)`.

We say that $R = r_1 \cdots r_m$ is in the class STR of *exact string patterns* (or *string patterns*) if every component r_i is a character in Σ such as R_1.

3. Proposed Architecture

In this section, we present our dynamically reconfigurable bit-parallel NFA architecture, Dynamic BP-NFA, based on NFA-simulation using bit-parallel pattern matching.

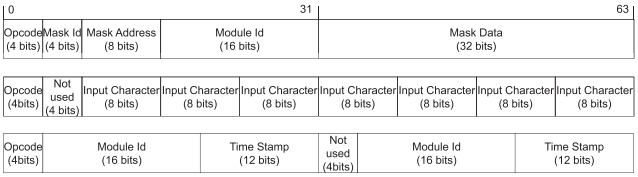

Fig. 2 The formats of the general I/O packets, the preprocessing input, run-time input, and run-time output packets from the top to the bottom.

3.1 Top-Level Architecture

In Fig. 1, we show the top-level architecture of our pattern matching hardware on FPGA. The hardware consists of the following submodules: an *input decoder*, an *output encoder*, and a collection of *pattern matching modules* (*PMM*). 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 and are classified into four types according to their `Opcode` field: *no-operation*, *preprocessing input*, *run-time input*, and *run-time output* packets. In Fig. 2, we show the preprocessing input, run-time input, and run-time output packet. The no-operation packet has only `Opcode` field and does nothing. The hardware runs in two different modes: *preprocessing* and *run-time* modes.

Preprocessing mode. In this mode, the hardware loads the description of input patterns with a preprocessing packet. A preprocessing packet has `Opcode` field for the packet type, `Mask Data` field for the data, and also has the three fields below to specify the location of a bit-mask to deliver: `Module Id` to specify the target PMM, `Mask Id` to specify a register or a block RAM, and `Mask Address` to specify the line (in the case of a block RAM only).

Run-time mode. In this mode, the hardware receives an input character, makes a state transition for the target NFA by a fixed circuitry, detects matches, and emits a matching information by receiving and sending run-time packets. A run-time input packet has `Opcode` field for its type and a sequence of `Input Character` fields for input characters. At each clock, one input character is fed to all PMMs. A run-time output packet has `Opcode` field for its type, a sequence of pairs of `Module Id` and `Time Stamp` fields for matching information. If a PMM detects the matching, then it sends matching information to the output encoder. Then, the output encoder packs a collection of matching information into a run-time output packet and send it to a host PC.

4. Pattern Matching Module

A *pattern matching module*, *PMM*, is a core of our pattern matching hardware and is responsible for NFA-simulation of a specified input pattern with fixed length $w \geq 1$. In what follows, we assume that $w = 32$, where w is actually the bit-length of registers in an underlying hardware.

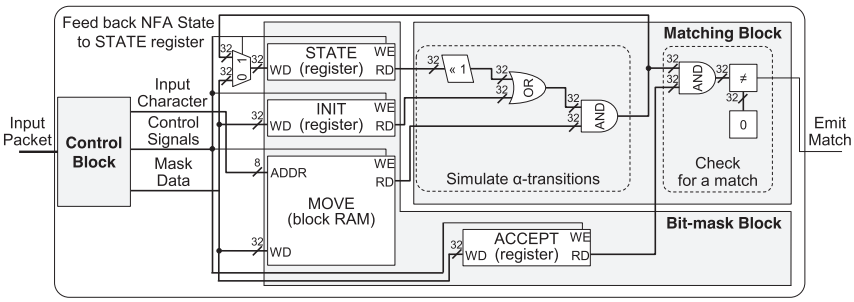

Fig. 3 The circuit of a pattern matching module for exact string patterns with bit-length $w = 32$.

4.1 Components of a Module

In the middle of Fig. 1, we show a single unit of a pattern matching module, PMM. A pattern matching module consists of three subunits: the *control block* for loading of bit-masks and input characters, the *bit-mask block* for storing NFA information, and the *matching block* as a control logic for NFA-simulation. We give assumptions on the FPGA device for describing PMMs. An input alphabet is the set $\Sigma = \{0, \ldots, 255\}$ of 8 bit characters. Each PMM has a number of registers and block RAMs of the same bit-length $w = 32$, which typically varies from 32 to 128 (bits). For each bit-mask, LSB (MSB, resp.) comes at the left end (at the right end, resp.).

A basic idea of bit-parallel pattern matching approach is to firstly transform a given extended pattern into a special NFA having linear shape, secondly to build a set of bit-masks from the transition relation of the NFA, and finally to make NFA-simulation on the bit-masks using by a fixed control logic designed to the target class of patterns. In the followings, we give the detailed description of our architecture step by step starting from simpler to more complex patterns.

4.2 NFA-Simulation: Exact String Pattern

First, we start with the construction of PMM for the class STR of exact string patterns based on the SHIFT-AND method [2], [16], [23]. We show in Fig. 3 the circuit of a PMM for STR. It consists of the definition of a set of bit-masks and a control logic for NFA-simulation. An *exact string pattern* is just a string $R = r_1 \cdots r_m$ of m characters, where $m \leq w$ and $r_i = a_i \in \Sigma$ is a character for every $i = 1, \ldots, m$.

Construction of NFA. First, we build the *exact string pattern NFA* $N_R = N(R)$ for an exact string pattern R as follows. First, we start with the state 0, and add the initial self-loop labeled with Σ to the state. Then, for every $i = 1, \ldots, m$, we add to the NFA N_R the new state i and the edge $e_i = (i-1, r_i, i)$ directed from the previous state $i-1$ to the current state i labeled with $r_i = a_i \in \Sigma$. The resulting NFA N_R consists only of the backbone of m edges labeled with characters and the initial self-loop. For example, we show in Fig. 4 the exact string pattern NFA $N_1 = N(R_1)$ corresponding to the exact string pattern $R_1 = \texttt{ABABBC}$.

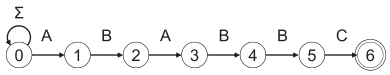

Fig. 4 The exact string pattern NFA of $R_1 = \texttt{ABABBC}$.

Precisely speaking, the NFA is given by the tuple $N_R = (\Sigma, Q, \delta, q_0, q_f)$, which has the state set $Q = \{0, 1, \ldots, m\}$, the initial state $q_0 = 0$, the final state $q_f = m$. The transition relation $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the set of directed edges, $\{(i-1, a_i, i) \mid i = 1, \ldots, m\}$, called the *backbone* of N_R, where ε is the empty string. The state 0 has the self-loop with labeled with Σ. In what follows, we call by *α-transitions* and *ε-transitions* state transitions by edges labeled with $c \in \Sigma$ and ε, respectively.

Construction of bit-masks. To simulate the exact string pattern NFA $N_R = N(R)$, we use w-bit masks $INIT$, $ACCEPT$ and an array $MOVE[c] \in \{0,1\}^w$ $(c \in \Sigma)$ of bit-masks defined as follows, where a mask stores all states $\{1, \ldots, m\}$ but state 0:

- $INIT$ is the w-bit mask that sets 1 at the bit-position for the state 1. That is, $INIT[i] = 1$ if and only if $i = 1$.
- $ACCEPT$ is the w-bit mask that sets 1 at the bit-position for the final state m. That is, $ACCEPT[i] = 1$ if and only if $i = m$.
- $MOVE[c]$ is the w-bit mask that indicates all bit-positions of backbones labeled with a character c in R. That is, $MOVE[c][i] = 1$ if and only if the state i has an incoming edge labeled with $c \in \Sigma$, i.e., $r_i = c$.

Note that we can easily extend the array $(MOVE[c])_{c \in \Sigma}$ to deal with character classes $\beta \subseteq \Sigma$, i.e., union of characters as follows: $MOVE[c]$ is the w-bit mask that indicates all bit-positions of backbones labeled with a character class $\beta \subseteq \Sigma$ with $c \in \beta$ in R.

We store the bit-masks $INIT$ and $ACCEPT$ in w-bit registers, and the array $(MOVE[c])_{c \in \Sigma}$ in a block RAM with a single read/write ports of $|\Sigma|$ entries with w bit-length.

Control logic for NFA-simulation. Based on the SHIFT-AND method [2], [16], [23], we finally give the control logic for NFA-simulation in the matching block as follows. First, the next code simulates the α-transitions, where $t \in \Sigma$ is the current character in an input text:

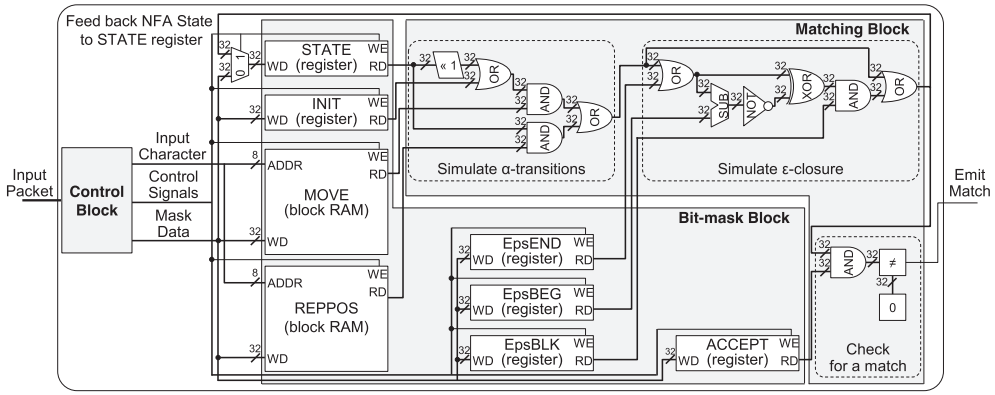

Fig. 5 The circuit of a pattern matching module for extended patterns with bit-length $w = 32$.

$$STATE \leftarrow ((STATE \ll 1) \mid INIT)\ \&\ MOVE[t];$$

Then, the following code checks for a match:

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

By the above construction, we can implement the control logic for NFA-simulation by a circuit shown in Fig. 3 by using five w-bit Boolean operations, three w-bit registers, and one block RAM with $|\Sigma|$ entries of w bit-length.

4.3 NFA-Simulation: Extended Pattern

Next, we show the construction of PMM for the class EXT of extended patterns based on the Extended SHIFT-AND method [16], [17]. We show in Fig. 5 the circuit of a PMM 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) $r_i = \beta$, (ii) $r_i = \beta?$, and (iii) $r_i = \beta^*$, (iv) $r_i = \beta^+$, and (v) $r_i = \beta\{x, y\}$, where $\beta \subseteq \Sigma$. We expand all occurrences of bounded repeats $r_i = \beta\{x, y\}$ of type (v) in R by using the equivalence $\beta\{x, y\} \equiv (\beta?)^{y-x}\beta^x$, where $x \le y$. Let $\textsc{Expand}(R) = r_1 \cdots r_m$ be the resulting extended pattern of m components, called the *expanded form* of R, where $|R| \le m \le w$. By construction, $\textsc{Expand}(R)$ contains no occurrences of components of type (v). Let $\mathcal{I} = \{1, \ldots, m\}$ be the set of all component indexes of $\textsc{Expand}(R)$. Then, we assign the unique numbers $1, \ldots, m$, called the *bit-positions*, to indexes in $\mathcal{I}$.

For example, we show in Fig. 6 the bit-position assignment for $R_2 = \texttt{[AB]}^+\texttt{B}.\{1, 3\}\texttt{[AC]}?.^*\texttt{C}$ consisting of six components. By replacing the bounded gap $.\{1, 3\}$ with $(.?)(.?)(.)$, we obtain its expanded form $\textsc{Expand}(R_2) = (\texttt{[AB]}^+)(\texttt{B})(.?)(.?)(.)(\texttt{[AC]}?)(.^*)(\texttt{C})$ consisting eight components with assigned bit-positions from 1 to 8.

Construction of NFA. Then, we build 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 $m \ge 1$ and w be an positive integer larger than or equal to m. By construction, we can assume that $\textsc{Expand}(R)$ contains components of only type (i)–(iv). First, we start with the state 0, and add the initial self-loop labeled with Σ to the state similarly to the construction of an exact string pattern NFA. Then, for every $i = 1, \ldots, m$, we add to the NFA N_R the new state i and the α-transitions and ε-transitions related to the state i according to the type of the i-th component r_i with matrix $\beta \subseteq \Sigma$ as follows:

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

Bit-position i	1	2	3	4	5	6	7	8
R_2	$\texttt{[AB]}^+$	B	.{1, 3}			[AC]?	.*	C
$\textsc{Expand}(R_2)$	$\texttt{[AB]}^+$	B	.?	.?	.	[AC]?	.*	C

Fig. 6 The bit-position assignment for $R_2 = \texttt{[AB]}^+\texttt{B}.\{1, 3\}\texttt{[AC]}?.^*\texttt{C}$ and its expanded form $\textsc{Expand}(R_2)$.

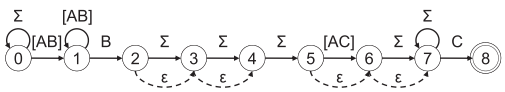

Fig. 7 The extended pattern NFA of $R_2 = \texttt{[AB]}^+\texttt{B}.\{1, 3\}\texttt{[AC]}?.^*\texttt{C}$.

We define an *ε-block* of the expanded form $\textsc{Expand}(R) = r_1 \cdots r_m$ by the set $B = \{i, \ldots, j\} \subseteq \mathcal{I}$ of the component indexes for a maximal consecutive subsequence $r_i \cdots r_j$ $(1 \le i \le j \le m)$ in $\textsc{Expand}(R)$, where r_k is either $r_k = \beta_k?$ or $r_k = \beta_k^*$ for every $i \le k \le j$. We can easily obtain all ε-blocks by scanning the components of $\textsc{Expand}(R)$ and finding such maximal consecutive subsequences. Let $\{B_1, \ldots, B_h\}$ $(h \ge 0)$ be the set of all ε-blocks of $\textsc{Expand}(R)$.

For example, we show in Fig. 7 the extended pattern NFA $N_2 = N(R_2)$ corresponding to $\textsc{Expand}(R_2)$. Then, $\textsc{Expand}(R_2)$ has two ε-blocks $B_1 = \{3, 4\}$ and $B_2 = \{6, 7\}$ corresponding to $r_3 r_4 = (.?)(.?)$ and $r_6 r_7 = (\texttt{[AC]}?)(.^*)$, respectively.

Construction of bit-masks. To simulate an extended pattern NFA $N_R = N(R)$, we use w-bit masks $EpsBEG$, $EpsEND$, $EpsBLK$, and the array $REPPOS[c] \in \{0,1\}^w$ of bit-masks in addition to the bit-masks $INIT$, $ACCEPT$, and $MOVE[c] \in \{0,1\}^w$ ($c \in \Sigma$) defined in the previous subsection.

- $EpsBEG$ is the w-bit mask that sets 1 at the previous bit-position of the lowest bit-position of every ε-block. That is, $EpsBEG[i] = 1$ if and only if $i = \min(B_k) - 1$ for some ε-block B_k.
- $EpsEND$ is the w-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 w-bit mask that sets 1s at all bit-positions of every ε-block. That is, $EpsBLK[i] = 1$ if and only if i is contained by some ε-block B_k.
- $REPPOS[c]$ is the w-bit mask that indicates all bit-positions of self-loops labeled with a character class $\beta \subseteq \Sigma$ with $c \in \beta$ in $\textsc{Expand}(R)$. That is, $REPPOS[c][i] = 1$ if and only if the state i has a self-loop labeled with $c \in \beta$, or equivalently, either $r_i = \beta^*$ or $r_i = \beta^+$ with $c \in \beta$.

For example, we show in Fig. 8 the bit-masks for R_2. As in the previous case, we store the bit-masks $INIT$, $ACCEPT$, $EpsBEG$, $EpsEND$, and $EpsBLK$ in w-bit registers, and the arrays $(MOVE[c])_{c\in\Sigma}$ and $(REPPOS[c])_{c\in\Sigma}$ in block RAMs.

Control logic for NFA-simulation. Based on the Extended SHIFT-AND method [16], we finally give the control logic for NFA-simulation in the matching block according to the codes in [16] as follows. First, the next code initializes the state mask at line (1), simulates the α-transitions at line (2), and simulates the α-transitions by self-loops at line (3), where $t \in \Sigma$ is the current character in an input text:

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

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	1	1	0
$MOVE$[B]	1	1	1	1	1	0	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	0	1	1	0	1	1	0

Fig. 8 The set of bit-masks for R_2 = [AB]$^+$B.{1, 3}[AC]?.*C on alphabet Σ = {A, B, C}, where the symbol '%' denotes any character not in Σ.

Then, the sequence of the following codes simulate the ε-transitions with the state mask:

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

The meaning of the above codes is explained as follows. At line (4), we turn on the highest bit (the *end bit*) of each ε-block in $STATE$, and set it to $HIGH$. At line (5), we invert all bits lower than or equal to the lowest 1 bit of all and the previous bits of each ε-block in $HIGH$ and set it to LOW. At line (6), the mask ($EpsBLK$ & (($\sim LOW$) $\oplus$ $HIGH$)) has 1s at all bit-positions properly higher than the lowest 1 bit of all and the previous bits of each ε-block in $STATE$. Finally, we add the change to $STATE$ at line (7). The code $STATE$ & $ACCEPT$ that checks for a match is same as exact pattern matching. In Fig. 9, we show an example of NFA-simulation by the set of bit-masks for R_2 = [AB]$^+$B.{1, 3}[AC]?.*C on an input text T = ABAABC. In Fig. 9, we show the status of the mask $STATE$ after the update in each cycle i ($1 \le i \le 6$). The output $EmitMatch$ of PMM is the value at the bit-position 8 of $STATE$.

By the above construction, we can implement the control logic for NFA-simulation by a circuit shown in Fig. 5 by using twelve w-bit Boolean operations, one w-bit subtraction, six w-bit registers, and two block RAMs with $|\Sigma|$ entries of w bit-length.

Theorem 1. *For the class* EXT *of extended patterns, our hardware consumes one input character per clock regardless of the content of the input text T. Furthermore, its combinatorial circuit for state update in Fig. 5, excluding registers and RAMs, has $O(\log w)$ depth and $O(w^3)$ gates, where w is the length of a register.*

Proof. The circuit in Fig. 5 contains one w-bit adder and constant number of w-bit bitwise Boolean gates, an w-bit multiplexer and an w-bit comparator. It is well known that an w-bit carry look-ahead adder can be implemented in $O(\log w)$ depth using $O(w^3)$ gates. Since the other w-bit gates can be implemented in constant depth and $O(w)$ 1-bit gates, we have the claimed complexities. Since any cycle on the data paths contains at most one register or RAM, the result is prove. □

Cycle i	Input character t_i	$STATE$ after update in cycle i							
		1	2	3	4	5	6	7	8
1	A	1	0	0	0	0	0	0	0
2	B	1	1	1	1	0	0	0	0
3	A	1	0	1	1	1	1	1	0
4	A	1	0	0	1	1	1	1	0
5	B	1	1	1	1	1	1	1	0
6	C	0	0	1	1	1	1	1	1

Fig. 9 An example of extended pattern matching, given an extended pattern R_2 = [AB]$^+$B.{1, 3}[AC]?.*C and an input text T = ABAABC.

5. Experimental Results

To evaluate the time and area complexities, we implemented our hardwares in Verilog HDL for both classes of exact string patterns STR (Sect. 4.2) and extended patterns EXT (Sect. 4.3), where the register length w is set to $w = 32$ and the arrays $MOVE$ and $REPPOS$ are implemented in block RAMs. We targeted the Virtex-5 LX330 with −2 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. All experiments were run in a PC (Intel Core2 Duo CPU, 2.40 GHz, 4.00 GB memory, Windows Vista).

5.1 Results on Our Dynamically Reconfigurable Hardwares

We give the experimental results for our pattern matching modules, PMM, in our Dynamic BP-NFA. In Table 1 and Table 2, we show the summaries of parameters for single and multiple PMMs, respectively.

Performance evaluation. The maximum frequencies of one PMM were 331 MHz and 184 MHz after place-and-route (418 MHz and 235 MHz after synthesis, resp.) for STR and EXT, respectively. For the time complexity in run-time, we estimated the throughput of matching by Throughput = Frequency × 8 (bit/sec) since our hardwares consume one character (8 bits) per clock. Thus, the throughputs were 2.6 Gbps and 1.5 Gbps after place-and-route (3.3 Gbps and 1.9 Gbps after synthesis, resp.) for STR and EXT, respectively. In what follows, the frequencies and throughputs of our Dynamic BP-NFA are those after place-and-route. The maximum frequencies and throughputs of PMMs for both STR and EXT is almost constant regardless of the number N of PMMs since our PMMs are independent of each other in our architecture. However, as indicated in [4], the overall performance of our system, including I/O, decreased as the N number of PMMs increased because of fanout delays of the control block feeding input characters to PMMs.

Resource usage. As shown in Table 1, for STR, one PMM used 52 slices and 1 block RAM (256 = 1×256 lines), and for EXT, one PMM used 140 slices and 2 block RAMs (512 = 2×256 lines). As shown in Table 2, we could implement up to 256 PMMs for STR (8,192 total characters) and up to 128 PMMs for EXT (4,096 total characters), where each PMM deals with a single pattern. For EXT, we used 12,124 slices total and 256 block RAMs. Consequently, the usage of block RAMs was 89%, while the usage of slices was only 23%. 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 slices. The number #Slice of slices was proportional to the number N of PMMs as expected.

Reconfiguration time. We evaluate the reconfiguration time of a pattern matching hardware, which is the time required by the hardware to load a description of input patterns. Therefore, the reconfiguration time of one PMM was estimated by Load Time = (#Reg + #BL)/Frequency (sec) to load a set of bit-masks. From the result of Table 1, one PMM took 0.782 μsec for STR and 2.82 μsec for EXT. to load an input pattern. Consequently, PMMs took 0.208 msec for STR and 0.377 msec for EXT to load all 256 and 128 patterns, respectively.

5.2 Comparison against Our Static Compilation Hardware

We compared our Dynamic BP-NFA for the class STR of exact string patterns against a static compilation hardware for STR, called the Static BP-NFA [11], which was implemented and evaluated on Virtex-5 LX50 with -1 speed grade, which has 7,200 slices. Below, we compare our Dynamic BP-NFA for STR against the Static BP-NFA in terms of performance evaluation, resource usage, and reconfiguration time.

Performance evaluation. As shown in Table 1, our Dynamic BP-NFA for STR achieved the frequency of 319 MHz and the throughput of 2.6 Gbps for 256 PMMs. On the other hand, the Static BP-NFA achieved the frequency of 216 MHz and the throughput of 1.7 Gbps for

Table 1 Summary of parameters of a single pattern matching module, where we assume $|\Sigma| = 256$. Class is the target class, #Op, #Add, #Reg, #BL, and #Slice are the numbers of 32-bit Boolean operations, 32-bit integer additions, registers, block RAM lines, and slices per pattern matching module, and Load Time is the loading time of an input pattern, respectively.

Class	#Op	#Add	#Reg	#BL	#Slice	Frequency	Throughput	Load Time
STR	5	0	3	256	52	331 MHz	2.6 Gbps	0.782 μsec
EXT	12	1	6	512	140	184 MHz	1.5 Gbps	2.82 μsec

Table 2 Summary of parameters of multiple pattern matching modules, where we assume $|\Sigma| = 256$. Class is the target class, #Modules and #BRAMs are the numbers of modules and block RAMs, #Chars Total is the total size of input patterns, and Load Time Total is the total loading time of input patterns, respectively.

Class	#Modules	#BRAMs	#Chars Total	Frequency	Throughput	Load Time Total
STR	256	256	8,192	319 MHz	2.6 Gbps	0.208 msec
EXT	128	256	4,096	176 MHz	1.4 Gbps	0.377 msec

Table 3 Results on comparisons of regular expression matching hardwares based on various dynamically reconfigurable architectures, where Class is the target class, Throughput is the calibrated and original throughputs (the original one is in parentheses), bRAM/char is the number of bytes used in block RAMs per character, LC/char is the number of logic cells used per character, and #Chars Total is the total size of input patterns.

Design	Class	Device	Throughput	bRAM/char	LC/char	#Chars Total
Dynamic BP-NFA for STR	STR	Virtex-5 LX330	2.6 (2.6) Gbps	32 bytes/char	3.2 LC/char	8192
Dynamic BP-NFA for EXT	EXT	Virtex-5 LX330	1.4 (1.4) Gbps	64 bytes/char	11.9 LC/char	4096
KMP-based hardware [4]	STR	Virtex-II Pro	3.6 (1.8) Gbps	4 bytes/char	3.2 LC/char	3200
Bitsplit-based hardware [8]	STR	Virtex-4 FX100	2.2 (1.6) Gbps	46 bytes/char	1.4 LC/char	16715
RegExp Controller hardware [3]	REG	Virtex-4 FX100	1.9 (1.4) Gbps	46 bytes/char	2.56 LC/char	16715

300 PMMs [11]. For a fair comparison of performance, we also implemented our Dynamic BP-NFA for STR with the same speed grade as the Static BP-NFA, i.e., -1, and achieved the throughput of 2.2 Gbps. Therefore, our Dynamic BP-NFA seems to be comparable to the Static BP-NFA in their performances though our dynamic hardware allows quick on-the-fly loading of input patterns.

Resource usage. Our Dynamic BP-NFA for STR totally used 6,500 slices for 256 PMMs. On the other hand, the Static BP-NFA used much less resources than our Dynamic BP-NFA. We could implement up to 1,500 PMMs (around 20 K total characters) using 7,200 slices and no block RAM, 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 [11].

Reconfiguration time. As shown in Table 1, our Dynamic BP-NFA for STR required the reconfiguration time of 0.208 msec for 256 PMMs. On the other hand, the Static BP-NFA has no such estimation formula. Therefore, we estimated the reconfiguration 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 [11]. Hence, our dynamic hardware is 10^6 times faster than the static one in reconfiguration time.

5.3 Comparison against Other Dynamically Reconfigurable Hardwares for Regular Expression Matching

In Table 3, we compare our NFA-based hardware against the previous DFA-based dynamically reconfigurable hardwares [3], [4], [8].

For a fair comparison of performance, we should be careful to interpret the throughputs of the previous hardwares in the original papers [3], [4], [8] since five dynamically reconfigurable hardwares including ours were evaluated in different settings. In this paper, we calibrated each original throughput by a factor α determined from process scaling on CMOS technology that the target FPGA device was built on, where $\alpha = 130/65 = 2.00$ for the hardware by [4] and $\alpha = 90/65 = 1.38$ for the hardwares by [3], [8] since the hardware by [4] targeted 130-nm Virtex-II Pro device, the hardwares by [3], [8] 90-nm Virtex-4 FX100 device, and our hardwares 65-nm Virtex-5 LX330. In what follows, we compare the throughputs of our Dynamic BP-NFA to the calibrated ones of the previous hardwares [3], [4], [8].

Performance evaluation. For the class STR of exact string patterns, our Dynamic BP-NFA achieved the throughput of 2.6 Gbps that is slower than Baker *et al.*'s KMP-based hardware [4] and higher than Jung *et al.*'s Bitsplit-based hardware [8]. For more general classes, Baker *et al.*'s RegExp Controller hardware [3], which is a hybrid of DFA-simulation and microcontroller, has been the only dynamically reconfigurable hardware for a non-trivial subclass of the whole class REG of regular expressions so far. Our Dynamic BP-NFA achieved the throughput of 1.4 Gbps for the class EXT of extended patterns, while the hardware by [3] achieved the higher throughput for REG. An advantage of our Dynamic BP-NFA is that it has theoretical performance guarantee in the worst-case from Theorem 1 in Sect. 4.3 for EXT as well as STR regardless of the actual contents of an input text, while the hardware by [3] has no performance guarantee in the worst-case when a regular expression has many occurrences of its subpatterns in an input text as indicated in [3].

Resource usage. First, we evaluate the block RAM usages of the hardwares by the parameter bRAM/char, which is the number of bytes used in block RAMs per character. As indicated in [3], [8], the effective utilization of block RAMs becomes important to dynamically reconfigurable hardwares because of modern FPGA devices equipped with large number of block RAMs. From the result of Table 1, our Dynamic BP-NFA used $256 \times 4 = 1024$ bytes per PMM for STR and $512 \times 4 = 2048$ bytes per PMM for EXT, and hence the block RAM usages of our hardwares are $1024/32 = 32$ bytes/char for STR and $2048/32 = 64$ bytes/char for EXT. Therefore, the block RAM usages of our hardwares for both STR and EXT are comparable to those of the hardwares by [3], [8]. We note that the actual block RAM usage of our hardware depends on a given pattern. For example, in the case that a given patten includes no character class, our hardwares for STR and EXT have at most 32 and 64 entries in block RAMs. In this case, our hardwares use $32 \times 4 = 128$ bytes per PMM for STR and $64 \times 4 = 256$ bytes per PMM for EXT, and hence the block RAM usages of our hardwares are $128/32 = 4$ bytes/char for STR and $256/32 = 8$ bytes/char for EXT, where we require an encoder from input characters to addresses of block RAMs. Therefore, the actual block RAM usages of our hardwares for both STR and EXT are less than those of the hardwares by [3], [8].

Then, we evaluate the logic cell usage of the hardwares by the parameter LC/char, which is the number of logic cells used per character and estimated by LC/char = 4×#Slices Total/#Chars Total since Virtex-5 contains four look-up tables and flip-flops per slice. To implement the circuitry for NFA-simulation as shown in Fig. 5, our Dynamic BP-NFA for EXT required five times more logic cells than the hardware by [3].

6. Conclusion

In this paper, we presented a novel architecture, called the dynamically reconfigurable bit-parallel NFA architecture, Dynamic BP-NFA, for large-scale regular expression matching. For the class STR of exact string patterns and the class EXT of extended patterns, which are subclasses of regular expressions, this architecture allows dynamic loading as well as fast pattern matching of its input patterns based on NFA-simulation by bit-parallel pattern matching. Our Dynamic BP-NFA is the first dynamically reconfigurable architecture for string and regular expression matching that fulfills the three requirements of dynamic loading of patterns, worst-case performance guarantee, and extensibility to more general patterns. Experimental results showed that our Dynamic BP-NFA for both STR and EXT had comparable performance to the existing dynamically reconfigurable architectures. Hence, our architecture presents an efficient alternative to existing dynamically reconfigurable hardwares for regular expression matching.

As future work, it is an interesting problem to extend our architecture to more general classes of patterns such as XPath queries [9] and network and regular expressions [10]. There are some hardwares that achieve speed-up by multi-character state transitions [19], [24], [26]. It is a future research to improve such techniques by using bit-parallel technique in, e.g., [7]. Finally, implementation of our architecture on GPGPU will also be an interesting problem.

Acknowledgments

The authors would like to thank Makoto Haraguchi, Takeru Inoue, Akira Ishino, Satoshi Kamiya, Shinobu Nagayama, Masayuki Takeda, Koji Tsuda, Takeaki Uno, and Osamu Watanabe 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 Univ., 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," Proc. SIGMOD'08, pp.147–160, 2008.

[2] R. Baeza-Yates and G.H. Gonnet, "A new approach to text searching," CACM, vol.35, no.10, pp.74–82, 1992.

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

[4] Z.K. Baker and V.K. Prasanna, "Time and area efficient pattern matching on FPGAs," Proc. FPGA'04, pp.223–232, 2004.

[5] L. Brenna, A.J. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W.M. White, "Cayuga: A high-performance event processing engine," Proc. SIGMOD'07, pp.1100–1102, 2007.

[6] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J.W. Lockwood, "Deep packet inspection using parallel bloom filters," IEEE Micro, vol.24, no.1, pp.52–61, 2004.

[7] K. Fredriksson, "SHIFT-OR string matching with super-alphabets," Inf. Process. Lett., vol.87, no.4, pp.201–204, 2003.

[8] H.J. Jung, Z.K. Baker, and V.K. Prasanna, "Performance of FPGA implementation of bit-split architecture for intrusion detection systems," Proc. RAW'06, p.189, 2006.

[9] Y. Kaneta and H. Arimura, "Faster bit-parallel algorithms for unordered pseudo-tree matching and tree homeomorphism," Proc. IWOCA'10, pp.68–81, 2010.

[10] Y. Kaneta, S. Minato, and H. Arimura, "Fast bit-parallel matching for network and regular expressions," Proc. SPIRE'10, pp.372–384, 2010.

[11] Y. Kaneta, S. Yoshizawa, S. Minato, H. Arimura, and Y. Miyanaga, "Efficient multiple regular expression matching on FPGAs based on extended SHIFT-AND method," Proc. SASIMI'10, pp.401–406, 2010.

[12] Y. Kaneta, S. Yoshizawa, S. Minato, H. Arimura, and Y. Miyanaga, "Dynamic reconfigurable bit-parallel architecture for large-scale regular expression matching," Proc. FPT'10, pp.21–28, 2010.

[13] Y. Kawanaka, S. Wakabayashi, and S. Nagayama, "A systolic regular expression pattern matching engine and its application to network intrusion detection," Proc. FPT'08, pp.297–300, 2008.

[14] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," Proc. SIGCOMM'06, pp.339–350, 2006.

[15] H. Nakahara, T. Sasao, and M. Matsuura, "A regular expression matching circuit based on a decomposed automaton," Proc. ARC'11, pp.16–28, 2011.

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

[17] G. Navarro and M. Raffinot, "Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching," J. Computational Biology, vol.10, no.6, pp.903–923, 2003.

[18] PCRE, 2011. http://www.pcre.org/

[19] H. Roan, W. Hwang, and C.D. Lo, "SHIFT-OR circuit for efficient network intrusion detection pattern matching," Proc. FPL'06, pp.1–6, 2006.

[20] R. Sidhu and V.K. Prasanna, "Fast regular expression matching using FPGAs," Proc. FCCM'01, pp.227–238, 2001.

[21] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata," Proc. SIGCOMM'08, pp.207–218, 2008.

[22] Snort, 2011. http://www.snort.org/

[23] S. Wu and U. Manber, "Fast text searching: Allowing errors," CACM, vol.35, no.10, pp.83–91, 1992.

[24] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," Proc. FPL'08, pp.131–136, 2008.

[25] Y.E. Yang and V.K. Prasanna, "Memory-efficient pipelined architecture for large-scale string matching," Proc. FCCM'09, pp.104–111, 2009.

[26] Y.E. Yang, W. Jiang, and V.K. Prasanna, "Compact architecture for high-throughput regular expression matching," Proc. ANCS'08, pp.30–39, 2008.

Yusaku Kaneta received B.S. in Engineering from Hokkaido University in 2007. He received M.S. in Computer Science from Hokkaido University in 2009. He is currently a doctral student of Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University. His research interests include pattern matching, data stream processing, VLSI design, and the design and analysis of algorithms in these fields.

Shingo Yoshizawa received the B.E., M.E., and Ph.D. degrees from Hokkaido University, Japan in 2001, 2003 and 2005, respectively. He was an Assistant Professor in the Graduate School of Information Science and Technology, Hokkaido University from 2006 to 2012. He is currently an Associate Professor in Department of Electrical and Electronic Engineering, Kitami Institute of Technology. His research interests are speech processing, wireless communication, and VLSI architecture.

Shin-ichi Minato is a Professor at Graduate School of Information Science and Technology, Hokkaido University. He also serves the Research Director of ERATO MINATO Discrete Structure Manipulation System Project, executed by JST. He received the B.E., M.E., and D.E. degrees in Information Science from Kyoto University in 1988, 1990, and 1995, respectively. He had been working for NTT Laboratories since 1990 until 2004. He was a Visiting Scholar at Computer Science Department of Stanford University in 1997. He joined Hokkaido University as an Associate Professor in 2004, and has been a Professor since Oct. 2010. He published "Binary Decision Diagrams and Applications for VLSI CAD" (Kluwer,1995). He is a member of IEEE, IPSJ, and JSAI.

Hiroki Arimura received the B.S. degree in 1988 in Physics, the M.S. and the Dr.Sci. degrees in 1990 and 1994 in Information Systems from Kyushu University. From 1990 to 1996, he was a research associate, a lecturer, and an associate professor in Kyushu Institute of Technology, and from 1996 to 2004, he was an associate professor in Kyushu University. Since 2006, he has been a professor of Hokkaido University, Sapporo, Japan. He has also been an adjunctive researcher with PREST program "Sakigake" of Japanese Science and Technology Agency for 1999 to 2002. His research interests include data mining, computational learning theory, information retrieval, artificial intelligence, and the design and analysis of algorithms in these fields. He is a member of JSAI, IPSJ, and ACM.

Yoshikazu Miyanaga is a professor in Graduate School of Information Science and Technology, Hokkaido University. He is an associate editor of Journal of Signal Processing, RISP Japan (2005-present). He was a chair of Technical Group on Smart Info-Media System, IEICE (IEICE TG-SIS) (2004-2006) and he is now a member of the advisory committee, IEICE TG-SIS (2006-present). He is also a vice-President, IEICE Engineering Science (ES) Society. He is a vice-President, Asia-Pacific Signal and Information Processing Association (APSIPA). He was a distinguished lecture (DL) of IEEE CAS Society (2010-2011) and he is now a Board of Governor (BoG) of IEEE CAS Society (2011-present).