TCS-TR-A-10-45

TCS Technical Report

Dynamic Reconfigurable Bit-Parallel Architecture for Large-Scale Regular Expression Matching

by

Yusaku Kaneta, Shingo Yoshizawa, Shin-ichi Minato, Hiroki Arimura, and Yoshikazu Miyanaga

Division of Computer Science

Report Series A

June 29, 2010

Hokkaido University
Graduate School of
Information Science and Technology

Email: arim@ist.hokudai.ac.jp

Phone: +81-011-706-7680
Fax: +81-011-706-7680

Dynamic Reconfigurable Bit-Parallel Architecture for Large-Scale Regular Expression Matching

Yusaku Kaneta, Shingo Yoshizawa, Shin-ichi Minato, Hiroki Arimura, and Yoshikazu Miyanaga
Graduate School of Information Science and Technology, Hokkaido University
N14, W9, Sapporo 060-0814, Japan
{y-kaneta,minato,arim,miya}@ist.hokudai.ac.jp
yosizawa@csm.ist.hokudai.ac.jp

***Abstract*—In this paper, we propose a novel FPGA-based architecture for large-scale regular expression matching, called *dynamic reconfigurable bit-parallel NFA architecture* (dynamic BP-NFA) that allows dynamic reconfiguration of the patterns using bit-parallel NFA-simulation approach. This is the first dynamic reconfigurable FPGA-based hardware with guaranteed performance for the class of extended patterns, where a extended pattern is a restricted regular expression in linear form consisting of letters, classes of letters, don't cares, optional letters, bounded and unbounded length gaps and repeatable letters. The key of our architecture is the use of *bit-parallel pattern matching* approach that have been developed in string matching communities for the decades. 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. As compared with previous approaches of DFA-based dynamic reconfigurable architectures, experimental results show that the proposed architecture achieves higher throughput for the class of exact string patterns and comparable for the class of extended patterns.**

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* of *complex* regular expressions against *high-speed* data streams.

These problems are, however, quite CPU-intensive tasks and it is difficult for a software on CPU to efficiently process massive data streams real time in wire-speed. Therefore, researches on large-scale regular expression matching on reconfigurable hardwares such as FPGA have attracted much attention recently.

B. Related works

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], [9], [11], [12], [14], [15], [16]). Then, this approach is further classified into the static compilation approach and the dynamic reconfiguration approach.

In the static compilation approach ([11], [12], [14], [15], [16]), a set of input regular expressions are transformed into either deterministic finite automata (DFA) or non-deterministic finite automata (NFA), 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], [9]), a universal control logic is statically compiled into FPGA beforehand, a description of regular expressions is loaded to the FPGA as data in preprocessing time, and then simulated in run-time [3], [4], [9]. This approach is attractive in real world applications such as EPS and NIDS where reconfiguration of input patterns frequently occurs. However, classes of patterns that can be dealt with in this approach are still limited, it is a challenging task to design dynamically reconfigurable hardwares that efficiently run for wider classes of regular expressions.

C. Main result of this paper

In this paper, we propose a novel architecture based on simulation of NFAs tailored for large-scale regular expression matching on FPGA, called *dynamic reconfigurable bit-parallel architecture* (dynamic BP-NFA). The key of this architecture is the use of *bit-parallel pattern matching* approach developed in string matching communities since 1990 [2], [10], [13].

As a main result, we present a hardware based on this approach for a subclass EXT of regular expressions, called *extended patterns*, which are regular expressions in linear form consisting of letters α, wildcards '.', classes of letters $[ab\cdots]$, optional letters $\alpha?$, bounded and unbounded repeats $\alpha\{x, y\}$ and α^*. For example, $R = \mathtt{[AB]^{+}B.\{1,3\}[BC]?.^{*}C}$ is an example of extended patterns.

In our approach, input extended patterns are 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 bit. 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. As analysis, we show that this hardware correctly matches a given set of extended patterns against an input text consuming one input letter per clock regardless of the contents of the input.

Compared with the previous dynamic reconfigurable DFA-based approaches, the throughput 2.9 Gbps of our hardware is much higher for class STR and the throughput 1.6 Gbps is comparable for class EXT, while it uses comparable hardware resources in FPGA implementation. An advantage of our architecture is the worst performance guaranteed by the design compared with DFA-based architecture with micro controller [3]. Another advantage is the potential extensibility for more general pattern classes. For example, Kaneta *et al.* [7] recently extended the Extended SHIFT-AND method, used in this paper, for a more general class of acyclic regular expressions allowing union and Kleene-plus. Such method can be incorporated into our architecture by extending the construction of masks and a circuitry.

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 each pattern matching module. In Section V, we give experimental results, and in Section VI, 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 a integers $i \in \mathbf{N}$, we define by a^i a 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 $a \in \Sigma$, concatenating $R = R_1 \cdot R_2$, union $R = (R_1|R_2)$, and the Kleene-star $R = (R_1)^*$, where R_1 and R_2 are regular expressions [10]. For a regular expression $R \in$ 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 $R = r_1 \cdots r_m \in$ REG occurs at 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 of regular expressions REG is defined as follows. An input is an input *pattern set* $\mathcal{P} = \{\ (i, R_i)\,|\, i = 1, \ldots, N\ \}$ $(N \geq 1)$, 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 (p, i) such that p is an end position of a pattern R_i in T for all $p = 1, 2, \ldots$ and $i = 1, \ldots, N$.

B. 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, $\equiv$ means the notational equivalence.

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:

(1) A *letter* $r_i = a \in \Sigma$ is a component with the language by $L(a) = \{a\}$.
(2) A *don't care* $r_i = .$ is a component with the language $L(.) = \Sigma$. This matches any letter in Σ.
(3) A *class of letters* $r_i = \beta$ is a component, where $\beta \subseteq \Sigma$, with the language $L(\beta) = \beta$. As notation, we write $[ab \cdots]$ for $\beta = \{a, b, \ldots\}$. Note that a letter $a \in \Sigma$ and a don't care symbol '.' are a class of letters.
(4) An *optional letter* $r_i = \beta?$ is a component, where $\beta \subseteq \Sigma$ is a class of letters, 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 don't care '.' then r_i is called a *bounded length gap*.
(6) *Unbounded repeats* $r_i = \beta^*$ and $r_i = \beta^+$ are components, where $\beta \subseteq \Sigma$ is a class of letters, and $\beta^+ = \beta\beta^*$. If β is a don't care '.' then r_i is called a *variable length don't care* (VLDC).

For $R = r_1 \cdots r_m$, we define its language by $L(R) = L(r_1) \cdots L(r_m)$. If a component 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.*

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

We say that R is an *exact string patterns* (also called a *string pattern*), denoted by STR, if every component r_i of an extended patterns $R = r_1 \cdots r_m$ is a letter in Σ such as R_1.

III. Proposed Architecture

In this section, we present our dynamic reconfigurable bit-parallel architecture, dynamic BP-NFA, based on simulation of NFAs using bit-parallel pattern matching technique.

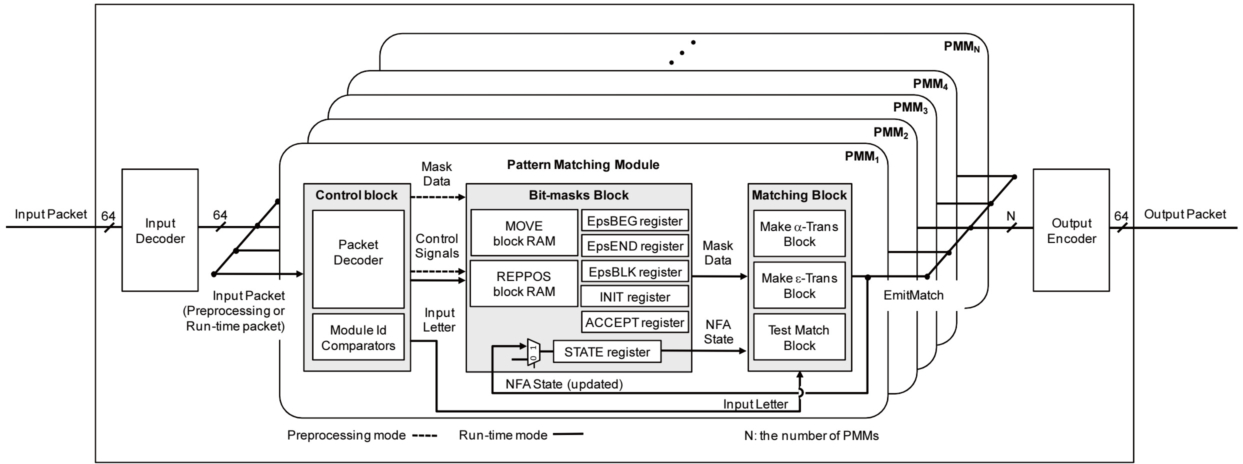

Fig. 1. The top-level architecture of our pattern matching hardware for extended patterns

A. Top-level architecture

In Fig. 1, we show the top-level architecture of our pattern matching hardware. The hardware consists of the input decoder, a collection of pattern matching modules (PMM, for short), and the output encoder as submodules.

The hardware runs with two different modes: the *preprocessing mode* and the *run-time mode*. It communicates the outside environment (a host PC) through a sequence of I/O packets of 64 bit length which are received and sent by the input decoder and output encoder.

B. Preprocessing and Run-time packets

I/O packets in all types have 64 bit length and start with the `Opcode` field of 4 bit length. I/O packets are classified into the following types according to `Opcode` field: *no-operation* (`0000`), *preprocessing* (`0001`), and *run-time* (`0010`). In Fig. 2, we show the bit layout of the preprocessing and run-time packets.

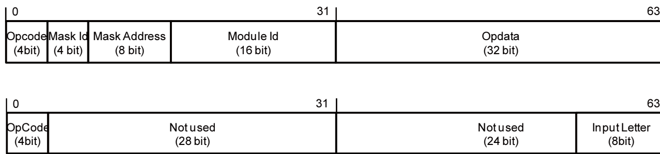

Fig. 2. The formats of the preprocessing and run-time packets

A *preprocessing packet* is an I/O packet with opcode `0001` (preprocessing) (Fig. 2). Then, the meaning of each field of a preprocessing packet is given by the following table.

A *run-time packet* is a packet with opcode `0010` (run-time) of the form shown in Fig. 2. Then, the first 32 bits contain the `Opcode` (4 bit) only and the remaining 32 bits contain the input letters in the `Opdata` field (32 bit). In present implementation, just 8 bit (one letter) in the `Opdata` field is used.

Received by the input decoder, an input packet is copies and delivered to all pattern matching modules. Each module

TABLE I
The meaning of fields in a preprocessing packet

Field	Length (bit)	Meaning
Opcode	4	The type of a packet
Mask Id	4	The name of the destination bit-mask
Mask Addr	8	An index of a line in a block RAM
Module Id	16	The id of the destination pattern matching module
Opdata	32	The content of a single bit-mask to deliver

PMM_i has its own decoder and decodes the fields of the received packet, where $1 \leq i \leq N$.

In the preprocessing mode, a host sends to the hardware only preprocessing packets. If its `ModuleId` field is identical to its own module id, namely i, then the module PMM_i stores the contents of the `Opdata` field at a register with name `MaskId` or a line `MaskAddress` of a block RAM with `MaskId` according to the values of `MaskId` and `MaskAddress`.

In the run-time mode, a host send to the hardware only run-time packet to feed input letters of an input stream. Then, the received run-time packet is delivered to all pattern matching modules, and an input letter as `Opdata` is extracted and processed by each module PMM_i.

C. Emission of match results

The *output-encoder* is a module consisting of a priority encoder [5] and a FIFO to keep the ids of matched patterns that report the match information to the outside environment using an *output packet*. An output packet is a packet of 64 bit length that contains two pairs of a `ModuleId` field (16 bit) and a field (16bit) for additional information such as the current time stamp. In the preprocessing mode, the output-encoder does nothing. In the run-time mode, if matching of a pattern P_i is detected, an `EmitMatch` signal (1 bit) is emitted from module PMM_i to the output-encoder. Then, the corresponding pattern id i in binary is pushed into the FIFO,

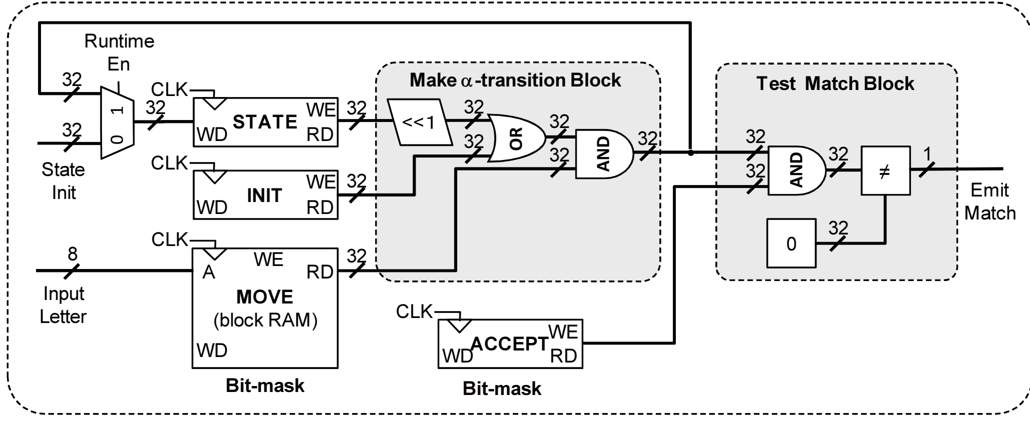

Fig. 3. The circuit of a pattern matching module for exact string patterns

packed into an output packet, and then sent to a host in each clock.

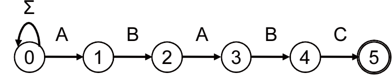

Fig. 4. The pattern NFA of a string pattern $R_1 = \mathtt{ABABC}$.

IV. Pattern matching module

A pattern matching module, PMM, is a core of our pattern matching hardware, and is responsible for simulation of a specified input pattern with fixed length, say, 32 letter length. In what follows, we assume registers and block RAM of 32 bit length.

A. 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: a decoding unit for delivery of masks (the *control block*), a memory unit for storing bit-masks (the *bit-masks block*), and a control logic for NFA-simulation (the *matching block*).

Before entering detailed description of a PMM, we give assumptions. The precise parameters setting depends on the actual class of patterns we target. We assume that an input alphabet is $\Sigma = \{0, \ldots, 255\}$, and thus, the width of an input letter is 8 (bit). We also assume that the register length is $L = 32$, which typically varies from 32 to 128 (bit). Then, each pattern matching module, PMM, has a number of registers and block RAMs of the same bit-length L. For each bit-masks, MSB (LSB, resp.) comes at the left end (at the right end, resp.).

A basic idea of bit-parallel pattern matching 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 functions 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 classes of patterns.

B. Simulation: Exact string pattern

We start with explaining the construction for the simplest class of exact string patterns according to SHIFT-AND method [2], [13] as follows. It consists of the definition of a set of bit-masks and a control logic for NFA-simulation. An *exact string pattern* (also called a *string pattern*) is just a string $P = r_1 \cdots r_m$ of m letters, where $m \leq L$ and $r_i = a_i \in \Sigma$ is a constant letter for every $i = 1, \ldots, m$. For example, we consider the string pattern $R_1 = \mathtt{ABABC}$ in the following explanation.

Construction of NFA. Then, the corresponding NFA $N_R = N(R)$ for a string pattern, called a *pattern NFA*, consists only of the backbone consisting only of m letter transitions. For example, we show in Fig. 4 the pattern NFA $N_1 = N(R_1)$ corresponding to the exact string pattern R_1. 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 letter edges $\{(i-1, a_i, i) \mid i = 1, \ldots, m\}$, called the *backbone* of N_R.

Construction of bit-masks. To simulate the pattern NFA for an exact string pattern, we use three types of L-bit masks $INIT$, $ACCEPT$ and an array $MOVE[a] \in \{0,1\}^L$ ($a \in \Sigma$) of bit-masks defined as follows:

- $INIT$ is the L-bit mask that sets 1 at the bit-position corresponding to the initial state. That is, $INIT[i] = 1$ if and only if $i = 0$.
- $ACCEPT$ is the L-bit mask that sets 1 at the bit-position corresponding to the final state. That is, $ACCEPT[i] = 1$ if and only $if i = m - 1$.
- $MOVE[a]$ is the L-bit mask that indicates all bit-positions of backbones labeled with a letter a in R. That

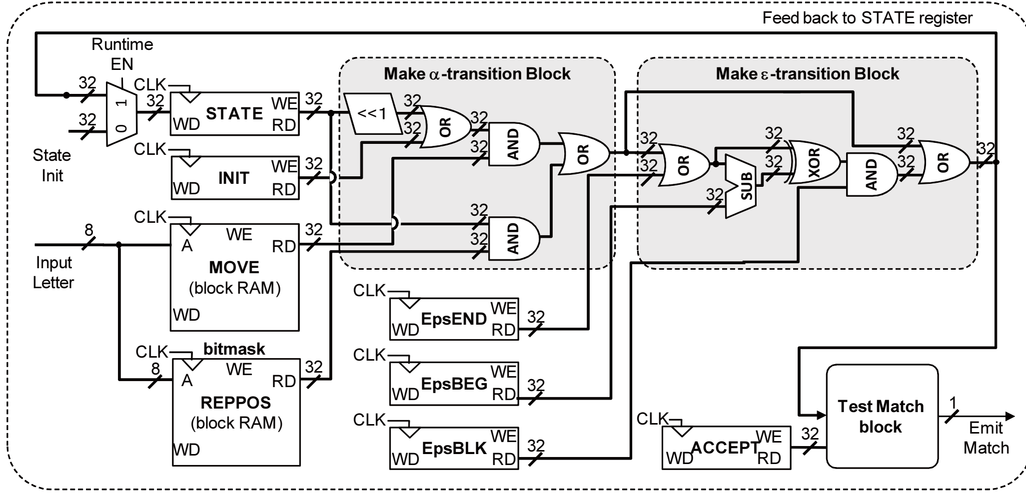

Fig. 5. The circuit of a pattern matching module for extended patterns

is, $MOVE[a][i] = 1$ if and only if the state i has a backbone labeled with $a \in \Sigma$, or equivalently, $r_i = a$.

We store the bit-masks $INIT$ and $ACCEPT$ in L-bit registers, and the array $(MOVE[a])_{a \in \Sigma}$ in a block RAM with Σ L-bit entries each having a single read and write ports.

Control logic for NFA-simulation. Based on SHIFT-AND method ([2], [10], [13]), the control logic in the matching block is, then, given by the following code, where $t \in \Sigma$ is the current input letter:

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

The acceptance test is given by the following code:

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

By the above construction, we can implement the control logic for the NFA-simulation by a circuit in Fig. 3 by using five L-bit Boolean operations, three L-bit registers, and one block RAMs with Σ L-bit entries each having a single read and write ports.

C. Simulation: Extended pattern

Now, we show the construction of PMM for extended pattern matching according to Extended SHIFT-AND methods (Chap. 4, Navarro and Raffinot [10]) as follows.

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 $\textsc{Expand}(R) = r_1 \cdots r_\ell$ be the resulting extended pattern consisting of m components, where $\ell \le L$. By construction, the resulting $\textsc{Expand}(R)$ contains no occurrences of components of type (v). Then, we assign the unique numbers $1, \cdots, \ell$, called the *bit-positions*, to all components $r_1, \ldots, r_\ell$.

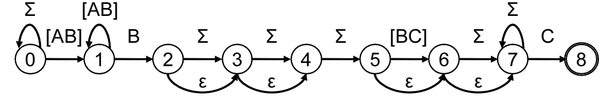

Fig. 6. The extended pattern NFA of $R_2 = \texttt{[AB]}^+\texttt{B.}\{1,3\}\texttt{[BC]?.*C}$.

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

Construction of NFA. Then, we obtain the pattern NFA $N(R)$ for R from the expanded version $\textsc{Expand}(R)$ as follows. Let $\textsc{Expand}(R) = r_1 \cdots r_m$ for some $m \ge 1$ and L be an positive integer larger than or equal to than m. L is acturally the length of regsiters 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 $N(R)$ the letter edges and ε-edges related to the state i according to the type of the i-th component r_i with matrix β as follows:

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

For example, we show in Fig. 6 the extended pattern NFA $N_1 = N(R_2)$ corresponding to the extended pattern $\textsc{Expand}(R_2)$.

For the expanded version $\textsc{Expand}(R)$ of an extended

pattern, an ε-*block* in $\text{EXPAND}(R)$ is a maximal consecutive subsequence $r_i \cdots r_j$, where a component r_i is $r_i = \beta_i?$ or $r_i = \beta_i^*$ for some $i \le j$. Let us denote by $B_1, \ldots, B_h$ $(h \ge 0)$ the ε-blocks of $\text{EXPAND}(R)$, where $B_j \subseteq \{1, \ldots, L\}$ is the set of bit-positions of the j-th ε-block. For example, $\text{EXPAND}(R_2)$ has two ε-blocks $B_1 = \{3, 4\}$ and $B_2 = \{6, 7\}$ corresponding to $r_3r_4 = (.?)(.?)$ and $r_6r_7 = ([\texttt{BC}]?)(.^*)$, resp.

Construction of bit-masks. To simulate an extended pattern NFA $N_R = N(R)$ for an extended pattern, we use L-bit masks $EpsBEG$, $EpsEND$, $EpsBLK$, and the array $REPPOS[a] \in \{0,1\}^L$ of bit-masks in addition to the bit-masks $INIT$, $ACCEPT$, and $MOVE[a] \in \{0,1\}^L$ $(a \in \Sigma)$ which are already defined in the previous subsection. For an L-bit mask, we identify each ε-block and the corresponding consecutive sequence of bit-positions in $\{1, \ldots, L\}$.

- $EpsBLK$ is the L-bit mask that sets 1's at all positions in every ε-block. That is, $EpsBLK[i] = 1$ if and only if i is contained by some ε-block B_k.
- $EpsBEG$ is the L-bit mask that sets 1 at the previous 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 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.
- $REPPOS[a]$ is the L-bit mask that indicates all bit-positions of self-loops labeled with a letter a in $\text{EXPAND}(R)$. That is, $REPPOS[a][i] = 1$ if and only if the state i has a self-loop labeled with $a \in \beta$, or equivalently, either $r_i = \beta^*$ or $r_i = \beta^+$ with $a \in \beta$.

In Fig. 8, we show an example of the bit-masks for $R_2 = [\texttt{AB}]^+\texttt{B}.\{1,3\}[\texttt{BC}]?.^*\texttt{C}$. As in the previous case, we store the bit-masks $INIT$, $ACCEPT$, $EpsBEG$, $EpsEND$, and $EpsBLK$ in L-bit registers, and the arrays $(MOVE[a])_{a\in\Sigma}$ and $(REPPOS[a])_{a\in\Sigma}$ in block RAMs.

Control logic for NFA-simulation. Based on bit-parallel method, we finally give the control logic for NFA-simulation. Fig. 5 shows the whole circuit of NFA-simulation for extended pattern matching. In the following, we give the computer code for the control logic this circuit for NFA-simulation, which is the modified version of the code in Navarro and Raffinot [10]. we note that $t = t_i \in \Sigma$ is the current input letter in an input text.

Firstly, the next code initializes the state mask at line (1), make a letter transition at line (2), and apply the letter transitions by self-loops at line (3):

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

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 code is explained as follows. At line (4), we turn on the highest bit (the *end bit*) of each ε-block of the mask $STATE$, 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 in $HIGH$ and set it to LOW. At line (6), the mask $(EpsBLK \;\&\; ((\sim LOW) \oplus HIGH))$ has bit 1 at all bit positions properly higher than the lowest 1 bit in $STATE$. Finally, we add the change to the mask $STATE$ at line (7). The acceptance test $STATE \;\&\; ACCEPT$ is same as exact pattern matching.

In Fig. 9, we show an example of NFA-simulation by the set of bit-masks for $R_2 = [\texttt{AB}]^+\texttt{B}.\{1,3\}[\texttt{BC}]?.^*\texttt{C}$ 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 $(1 \le i \le 6)$.

By the above construction, we can implement the control logic for the NFA-simulation by a circuit shown in Fig. 5 by using eleven L-bit Boolean operations, one L-bit subtraction, six L-bit registers, and two RAMs with Σ L-bit entries each having a single read and write ports.

Theorem 1. *For the class* EXT *of extended patterns, our hardware consumes one input letter per clock regardless of the content of the input text T. Furthermore, its combinatorial*

Bit-positions	1	2	3	4	5	6	7	8
R_2	$[\texttt{AB}]^+$	B	$.\{1,3\}$			$[\texttt{BC}]?$	$.^*$	C
$\text{EXPAND}(R_2)$	$[\texttt{AB}]^+$	B	$.?$	$.?$	$.$	$[\texttt{BC}]?$	$.^*$	C

Fig. 7. The bit-position assignment for $R_2 = [\texttt{AB}]^+\texttt{B}.\{1,3\}[\texttt{BC}]?.^*\texttt{C}$ and its expanded version $\text{EXPAND}(R_2)$.

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	0	1	1	0	1	1	0

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

TABLE II
Summary of parameters of the proposed pattern matching hardwares, where #Op, #Add, #Reg, #BL, and #Slice, are the numbers of 32 bit operations, 32 bit integer additions, registers, block RAM lines, resp., per PMM. #Patterns and #Chars Total are the number and the total size of input patterns, resp.

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	11	1	6	512	123	202 MHz	1.6 Gbps	0.328 msec	128	4,096 letters

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

circuit for state update in Fig. 5, excluding registers and RAMs, has $O(\log L)$ depth and $O(L^3)$ gates, where L is the length of a register.

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

V. Experimental results

To evaluate the time and area complexities, we implemented our pattern matching hardwares in Verilog HDL for both classes of exact string patterns STR (Sec. IV-B) and extended patterns EXT (Sec. IV-C), where the register length L is set to $L = 32$ and the arrays $MOVE$ and $REPPOS$ are implemented in block RAMs. We targeted the Virtex-5 LX330 with -1 speed grade, which has 51,840 slices and 288 block RAMs with 36 Kbits. We use the Xilinx ISE Design Suite 10.1 and Synopsys VCS development tools.

Summary of our hardwares. In Table. II, we show the summary of the experimental results of a single pattern matching module, PMM, in our pattern matching hardwares. The number of block RAM lines (#BL) is given by the number of block RAMs times $|\Sigma| = 256$.

Resource usage. For the area complexity, in the experiments, 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 uses 123 slices (787 logic cells) and two block RAMs ($512 = 2\times256$ lines are 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. In Fig. 10, we show the dependency of the number of slices against the number N of PMMs, where the number of slices is proportional to N.

Performance evaluation. The maximum frequencies of our hardwares were 363 MHz and 202 MHz, resp., for STR and EXT. For the time complexity in run-time, we estimate 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 bit) per clock. In Fig. 11, we observe that the throughputs for STR and EXT is constant regardless of N.

On dynamic reconfiguration, we estimate the total loading time of input patterns by Load Time Total $=$ #Patterns $\times$ (#Reg $+$ #BL)$/$Freq (sec). Thus, a PMM took 0.71 μsec and 2.56 μsec to load an input pattern, and consequently, the total loading times were 0.182 msec and 0.328 msec to load 256 and 128 patterns for PAT and EXT, resp. As comparison, we implemented a hardware in the static compilation approach based on SHIFT-AND method [2] for STR on Xilinx Virtex-5 LX50 with 7,200 slices. This hardware achieves frequency 216 MHz and throughput 1.7 Gbps using 2,925 slices and no block RAMs. As the loading time, the compilation required $4.27{\times}10^5$ msec, approximately seven minutes, for 300 patterns with total size 3,963 letters. Hence, the dynamic reconfiguration is 10^6 times faster than the static compilation in this case.

Comparison against other regular expression matching hardwares. In Table. III, we compared our hardware, dynamic BP-NFA, against the previous DFA-based dynamic reconfigurable hardwares [3], [4], [6]. For the class of exact string patterns, our hardware for STR achieves throughput of 2.9 Gbps higher than 1.8 Gbps by Baker *et al.*'s KMP-based hardware [4] and 1.6 Gbps by Jung *et al.*'s Bitsplit-based hardware [6]. For more complex classes of regular expressions, our hardware achieves 1.6 Gbps for the class EXT of extended patterns, while Baker *et al.*'s RegExp Controller hardware [3] achieves 1.4 Gbps for their target subclass of regular expressions, which will be explained below. On resource usage, it seems that our hardware uses ten times more logic cells (slices) than RegExp Controller hardware, while the former uses less block RAM lines than the latter.

Note that Baker *et al.*'s RegExp Controller hardware is a hybrid of DFA simulation and microcontroller [3] and has been only dynamic reconfigurable architecture for a subclass of regular expressions before ours, where each regular expression can be expressed with a small DFA and a few number of un-

TABLE III
Results comparisons of regular expression matching hardwares based on various dynamic reconfigurable architectures, where Class is the target class of input patterns, bRAM/char is the number of bytes used in block RAMs per letter, and Logic Cells/char is the number of logic cells used per letter. STR, EXT, and REG stand for the classes of exact string patterns, extended patterns, and regular expression, resp.

Design	Class	Device	Throughput	bRAM/char	Logic Cells/char	#Chars Total	Remarks
Dynamic BP-NFA for STR (ours)	STR	Virtex-5 LX300	2.9 Gbps	4 bytes/char	10.8	8192	54 slices
Dynamic BP-NFA for EXT (ours)	EXT	Virtex-5 LX300	1.6 Gbps	8 bytes/char	24.6	4096	123 slices
KMP-based hardware [4]	STR	Virtex-II Pro	1.8 Gbps	4 bytes/char	3.2	3200	NA
Bitsplit-based hardware [6]	STR	Virtex-4	1.6 Gbps	46 bytes/char	1.4	16715	NA
RegExp Controller hardware [3]	REG	Virtex-4 FX100	1.4 Gbps	46 bytes/char	2.56	16715	NA

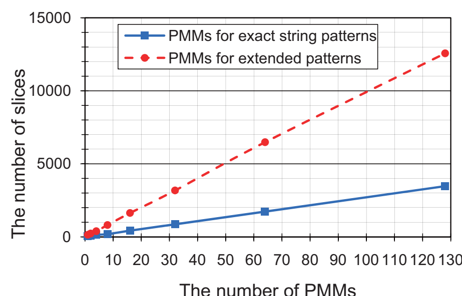

Fig. 10. Slice usages against the number of PMMs for exact string matching and extended pattern matching.

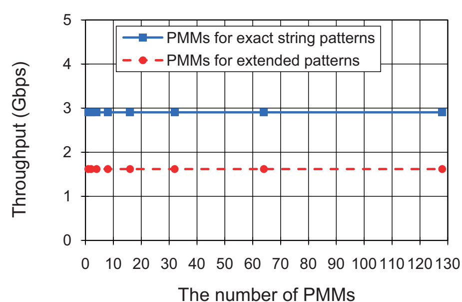

Fig. 11. Throughputs against the number of PMMs for exact string matching and extended pattern matching.

bounded and bounded repeats as delimiters. Although Baker *et al.*'s hardware achieves high throughput, it can be overflowed when many matches of short segments occur. On the contrary, our hardware, dynamic BP-NFA, consumes more logic, while it achieves comparable high throughput regardless of the type of input as shown in Theorem. 1. Hence, our hardware can be an alternative choice for dynamic reconfigurable hardwares for regular expression matching.

VI. Conclusion

In this paper, we presented a novel FPGA-based architecture, called the dynamic reconfigurable bit-parallel architecture, for large-scale regular expression matching. For the subclass EXT of regular expressions, this architecture provides dynamic re-configuration of the patterns using bit-parallel NFA simulation.

The use of block RAMs is attractive on modern FPGA devices equipped with large number of block RAMs [1] since it reduces not only the number of logic cells but also the compilation and place-and-route times.

In this paper, we considered the class of extended patterns. As future work, it is an interesting problem to extend our architecture for more general classes of patterns, such as acyclic regular expressions [7] and XPath queries [8].

References

[1] J. Agrawal, Y. Diao, D. Gyllstrom, 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. FPGA'04*, ACM, pp. 223–232, 2004.
[5] D. Harris and S. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann Publishers Inc., 2007.
[6] 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.
[7] Y. Kaneta, S. Minato, and H. Arimura. "An efficient matching algorithm for acyclic regular expressions with bounded depth". *TCS Technical Report A, DCS, Hokkaido University*, TCS-TR-A-10-40, Feb. 2010.
[8] Y. Kaneta and H. Arimura. "Faster bit-parallel algorithms for unordered pseudo-tree matching and tree homeomorphism". In *Proc. IWOCA'10*, 2010.
[9] Y. Kawanaka, S. Wakabayashi, and S. Nagayama. "A systolic regular expression pattern matching engine and its application to network intrusion detection" In *Proc. FPT'08*, pp. 297–300, 2008.
[10] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.
[11] H. Roan, W. Hwang, *et al.* "SHIFT-OR circuit for efficient network intrusion detection pattern matching". In *Proc. FPL'06*, pp. 1–6, 2006.
[12] R. Sidhu and V. K. Prasanna. "Fast regular expression matching using FPGAs". In *Proc. the 9th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pp. 227–238, 2001.
[13] S. Wu and U. Manber. "Fast text searching: allowing errors". *CACM*, 35(10), pp. 83–91, 1992.
[14] N. Yamagaki, R. Sidhu, and S. Kamiya. "High-speed regular expression matching engine using multi-character NFA". In *Proc. FPL'08*, pp. 131–136, 2008.
[15] 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.
[16] 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.

[1]In the latest version of Xilinx Virtex series, the highend model Virtex-7 910T, released in June 2010, has 1,800 block RAMs, while Virtex-5 240T, released in October 2006, has 516 block RAMs.