HOKKAIDO UNIVERSITY

| | |
|---|---|
| Title | Hardware Accelerators for Regular Expression Matching and Approximate String Matching |
| Author(s) | Wakabayashi, Shin'ichi; Nagayama, Shinobu; Kawanaka, Yosuke et al. |
| Description | APSIPA ASC 2009: Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference. 4-7 October 2009. Sapporo, Japan. Panel session: Panel Discussion 3. Hardware Software Co-research for Efficient Information Processing (7 October 2009). |
| Citation | Proceedings : APSIPA ASC 2009 : Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference |
| Issue Date | 2009-10-04 |
| Doc URL | https://hdl.handle.net/2115/39822 |
| Type | conference paper |
| File Information | ks_material03.pdf |

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

# Hardware Accelerators for Regular Expression Matching and Approximate String Matching

Shin'ichi Wakabayashi* Shinobu Nagayama† Yosuke Kawanaka Sadatoshi Mikami

Graduate School of Information Sciences, Hiroshima City University
3-4-1 Ozuka-higashi, Asaminami-ku, Hiroshima 731-3194 Japan
E-mail: *wakaba@hiroshima-cu.ac.jp †s_naga@hiroshima-cu.ac.jp

***Abstract*—This paper introduces hardware accelerators for regular expression matching and approximate string matching. The hardware for regular expression matching accepts a subclass of regular expressions, and achieves a high throughput string matching for a wide range of patterns. In addition, since the hardware is pattern-independent, we can update patterns immediately without reconfiguring the hardware. Therefore, it is useful for applications that require quick pattern updating, such as network intrusion detection. The hardware for approximate string matching calculates the edit distance as a degree of similarity between two strings at high speed. Therefore, it accelerates processing for text retrieval in database, analysis of DNA, protein sequences in bioinformatics, and so on.**

## I. Introduction

*String matching* is a problem to search for strings from the input text which match a given pattern. The problems can be classified into the *exact* string matching which is a problem to find strings *equivalent* to a pattern, and the *approximate* string matching which is a problem to find strings *similar* to a pattern [1], [5]. These problems occur in a broad range of applications, such as network intrusion detection, text retrieval in databases, analysis of DNA, and protein sequences in bioinformatics. Various algorithms for string matching have been extensively studied to shorten its computation time, and a number of research results have been presented in last 50 years [1], [20].

Since the advent of VLSI era, it has become possible to realize algorithms on VLSI circuits as hardware algorithms to drastically reduce the computation time to solve problems [16]. Many hardware algorithms have been proposed for various kinds of string matching [1]. For example, Foster, *et al.* [4] and Mukhopadhyay [19] proposed hardware algorithms for the string pattern matching problem. Kikuno, *et al.* [13] proposed a hardware algorithm for the longest common subsequence problem.

In this paper, we introduce two hardware algorithms for exact and approximate string matching problems. As for the exact string matching problem, we focus on a *regular expression matching* problem, in which a subclass of regular expressions is given as a pattern. And, as for the approximate string matching problem, we focus on a *string-to-string correction* problem which is a problem to calculate the edit distance as a degree of similarity between two strings [24].

The rest of this paper is organized as follows: Section II describes the hardware algorithm for regular expression matching, and shows its architecture. Section III describes the hardware algorithm for calculating the edit distance, and shows its architecture. Section IV gives some experimental results to show the effectiveness of our hardware algorithms. Finally, some concluding remarks are given in Section V.

## II. Regular Expression Matching

In this section, we introduce the hardware algorithm for regular expression matching and its architecture.

### A. Related Work on Exact String Matching

In this subsection, we quickly overview previous results on hardware algorithms for exact string matching problems. In 1970s and 1980s, hardware algorithms for simple character string matching [4], [19] have been presented. In addition, various hardware algorithms such as a special-purpose string matching hardware for database machines [15], content addressable memory (CAM) based methods [28], trie based methods [23], and hashing based methods [6] have been reported. These hardware algorithms can realize high speed string matching and immediate pattern updating. However, since in these algorithms, patterns used in matching are restricted only to simple character strings, their applications also can be restricted to ones that require only simple patterns.

On the other hand, regular expression matching, in which a regular expression is given as a pattern, can be used in a wider range of applications because a regular expression can encompass tens and hundreds of character strings [18], and can represent various patterns compactly. Particularly, in recent years, an applications of regular expression matching to network intrusion detection systems (NIDSs) has attracted much attention. In 2000s, FPGA implementation of regular expression matching for NIDSs has been widely investigated, and many research results have been reported [2], [21], [26].

As well known, any regular expression can be realized by either non-deterministic or deterministic finite automaton (NFA or DFA) [7], and NFA and DFA can be implemented as simple circuits. In this case, patterns are embedded as hardware circuits. Such pattern-specific circuits are high speed and compact for given patterns. Thus, most of the existing

researches have adopted a pattern-specific hardware implementation approach [2], [21], [26]. However, this pattern-specific approach has a major disadvantage. For those pattern-specific matching engines, if a pattern is changed, then a sequence of FPGA design and implementation processes (i.e., generating HDL code, logic synthesis, place and route, and generating a configuration data) should be performed again. This FPGA design sequence requires a fairly long time, sometimes, a few hours. This property of pattern-specific matching engines would be fatal for several applications like NIDSs, in which patterns are frequently updated, and quick response of matching results is strongly demanded.

To overcome this problem, pattern-independent regular expression matching engines which can update patterns immediately have been proposed [3], but they accept only a very small subclass of regular expressions. Although in many applications, the whole class of regular expressions is not required, a subclass large enough to describe patterns is desired. In this section, we introduce a pattern-independent regular expression matching engine which accepts a much larger subclass of regular expressions.

### B. A Subclass of Regular Expressions

In our hardware algorithm, a *pattern* is described as a regular expression $RE$ with some restrictions. In the following, we define a subclass of $RE$ [7].

Let $\sum = \{a_1, a_2, \ldots, a_s\}$ be a finite, nonempty set of symbols (or, characters). We call $\sum$ an *alphabet*. $|$, $\cdot$, and $*$ are the union, concatenation, and Kleene closure operators of regular expressions. The $*$ operator is of highest precedence, and the $\cdot$ operator is the next one. The $|$ operator has the lowest precedence among those three operators. The $\cdot$ operator is usually omitted, and for example, we denote "$abc$" instead of "$a \cdot b \cdot c$". ? is a special symbol not in $\sum$, and let $? = a_1|a_2|\ldots|a_s$.

Now, we define a subclass of regular expressions, which is used to specify a pattern in our algorithm. If expression $E_C$ is a symbol in $\sum \cup \{?\}$ or a concatenation of more than one symbols, then $E_C$ is called a *C-term*. Let $E_1, E_2 \ldots, E_k$, $k \geq 2$, be C-terms. If $E_U = E_1|E_2|\ldots|E_k$, then $E_U$ is called a *U-term*. Let $E$ be a U-term. If $E_K = (E)^*$, then $E_K$ is called a *UK-term*. If $E$ is a C-term, a U-term, a UK-term, or a concatenation of a finite number of C-, U-, and UK-terms, then $E$ is called *admissible*.

The problem of regular expression matching discussed in this section is the problem of finding all occurrences of substrings, which match a given pattern $P$, where $P$ is an admissible regular expression on alphabet $\sum$.

An example of admissible regular expression is shown as follows. Let $\sum = \{a, b, c, d, e\}$. Then, $abc(ac|de)^*(bd|ce)$ is an example of admissible regular expression.

For an admissible regular expression $P$, nesting of parentheses is not allowed. Thus, the language represented by a class of admissible regular expressions is strictly smaller than the normal regular language. However, there are many applications in the real world, in which admissible regular expressions are enough to specify patterns in regular expression matching.

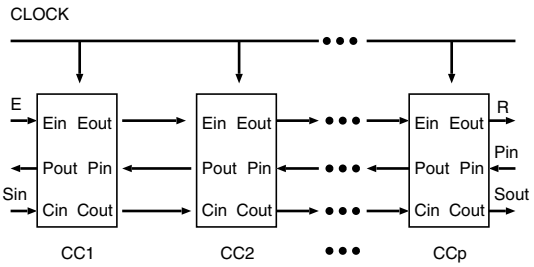


Fig. 1. Overview of the regular expression matching engine.

Given a pattern $P$, the length of $P$ is defined as the number of alphabet symbols included in $P$. That is, $*$, $\cdot$, and $|$ operators and parentheses in $P$ are not counted.

### C. Overview of the Architecture

Our string matching engine is constructed as a one-dimensional array of simple processing units called *comparison cell (CC)* as shown in Fig. 1. This architecture can be classified as a one-dimensional systolic array [14]. Assume that the length of given pattern $P$ is $p$, then there are $p$ CCs in the circuit.

The whole circuit is implemented as a synchronous circuit, and the behavior of the circuit is synchronized to one global clock signal. A CC is a processing unit, which stores one symbol in the pattern $P$, and one symbol in the text $T$ to be searched. The main function of CC is to compare a symbol in $P$ with a symbol in $T$. Pattern $P$ is input from the rightmost cell before starting string matching. Each symbol in $P$ is stored in each CC. A text to be retrieved is input from the leftmost cell, and string matching is performed in each CC in parallel and pipeline fashion. Details of the behavior of CC are described in the next subsection.

The matching algorithm supports two modes of string matching; one is the *anchor* mode, and the other is the *unanchor* mode [19]. The anchor mode is used to determine whether the whole input text $T$ matches with pattern $P$. In this mode, we set the enable signal $E_{in}$ of the leftmost cell $CC_1$ to *true* only when the first symbol in $T$ is input from the outside. If $T$ matches with $P$, then the output signal $E_{out}$ of the rightmost cell becomes *true* when the last symbol of $T$ is output from the rightmost cell.

On the other hand, the unanchor mode is used to determine whether the input text $T$ contains a substring, which matches with pattern $P$. In this mode, during the execution of the algorithm, we always set the enable signal $E_{in}$ of $CC_1$ to *true*. If $T$ includes a substring $S$, which matches with $P$, then the output signal $E_{out}$ becomes *true* when the $S$ is output from the rightmost cell.

### D. Comparison Cell

As mentioned, our regular expression matching engine consists of $p$ comparison cells. Fig. 2 shows the structure of

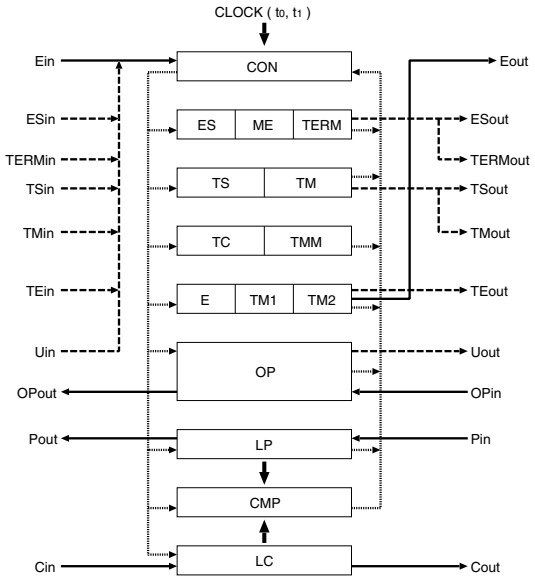


Fig. 2. Comparison cell.

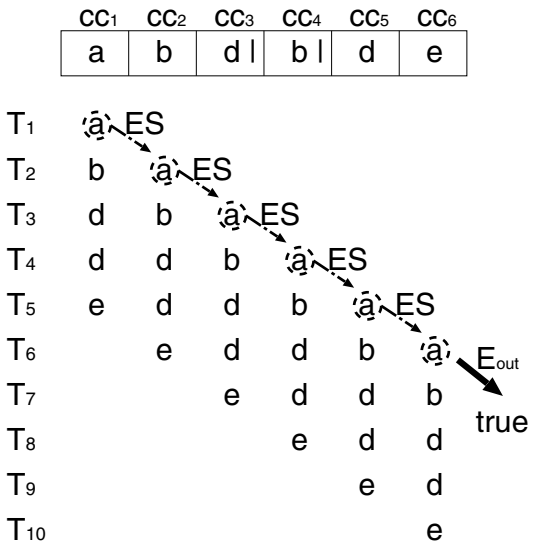


Fig. 3. Empty matching.

a comparison cell. It consists of the control circuit $CON$, flags $E$, $ES$, $TM1$, $TM2$, registers $OP$, $LP$, $LC$, $TERM$, $TS$, $TM$, counter $TC$, a comparator $CMP$, and FIFO buffer $TMM$.

$CON$ controls the overall behavior of the cell. The behavior of each cell is synchronized to a global clock signal. We assume a two-phase clocking scheme, and each clock cycle $T$ consists of a pair of two clock signals $(t_0, t_1)$, which is provided to each cell. When we want to describe $i$-th clock cycle, then we represent it as $T_i$.

$OP$ is a register which stores | operator and parentheses "(" and ")". "(" and its subsequent symbol in $P$ is stored in the same cell. ")" and "|" are stored in a cell having a symbol, whose subsequent symbol in $P$ is ")" or "|". $LP$ and $LC$ are registers to store symbols in the pattern and text, respectively. $CMP$ is a comparator which compares two symbols stored in registers $LP$ and $LC$. Other registers, a binary counter $TC$, and FIFO buffers $TMM$ are explained in the following subsection.

From the definition of patterns, a given pattern $P$ is a concatenation of C-, U-, and UK-terms. In the next subsection, first, a matching algorithm for UK-terms is described. Since matching algorithms for C- and U-terms are easily derived from the algorithm for UK-terms, their explanation will be given in the subsequent subsections.

### *E. Matching of a UK-term*

Let $P = (t_1|t_2|\ldots|t_k)^*$ be a UK-term, where each $t_i$ is a C-term. Assume that $P$ is given as a pattern to be retrieved. A matching algorithm for a UK-term consists of three types of matching; empty matching, tentative matching, and final matching. In the following, we explain those matching.

*1) Empty Matching:* From the definition of a UK-term, an empty input string is always matched with $P$. We call this matching *empty matching*. Empty matching is realized by the algorithm as follows. When the input signal $E_{in}$ of the leftmost cell becomes *true*, matching of an input text with pattern $P$ begins. For each cell, if its input $E_{in}$ becomes *true* in clock cycle $T_i$, this enable signal, denoted $ES$, is sent to its right neighbor cell in the next clock cycle $T_{i+1}$. As a result, the output signal $E_{out}$ of the rightmost cell becomes *true*, when the first symbol in the text string is output from the rightmost cell (see Fig. 3).

*2) Tentative Matching:* Assume that a UK-term $P = (t_1|t_2|\ldots|t_k)^*$ is given as an input pattern, where $t_1, t_2, \ldots, t_k$ are C-terms. When a given input text $T$ matches with $P$, $T$ can be divided into substrings, $T_1, T_2, \ldots, T_s$ such that $T = T_1T_2\ldots T_s$, and for each $T_i$, there is a C-term $t_j$ in $P$ and $T_i = t_j$. For example, let $P = (abd|b|de)^*$. Let $T = abdde$. Then, $P$ matches with $T$, and $T = T_1T_2$, $T_1 = abd, T_2 = de$.

From this observation, if you want to perform string matching for UK-terms correctly, we should realize the following two functions in the algorithm. The first function is to perform string matching for C-terms. For any substring $T_i$ in $T$, if there is a C-term $t_j$ in $P$ such that $T_i = t_j$, the algorithm should correctly decide that $T_i$ matches with $t_j$. In the following, we call this function *tentative matching* of $t_j$.

The second function of matching for UK-terms is to check whether the input text $T$ can be divided into substrings $T_1, T_2, \ldots, T_s$ such that $T = T_1T_2\ldots T_s$, and in tentative matching, it has been verified that for each $T_i$, there is a C-term $t_j$ in $P$ such that $T_i = t_j$. In the following, we call this function *final matching*. We will explain final matching in detail in the next subsection.

Tentative matching of a C-term $t_j$ is realized by a set of cells, which contain pattern symbols in $t_j$. For example, let $P = (abd|b|de)^*$. $P$ consists of three C-terms, $t_1 = abd$, $t_2 = b$, and $t_3 = de$. This pattern is set in the systolic algorithm as

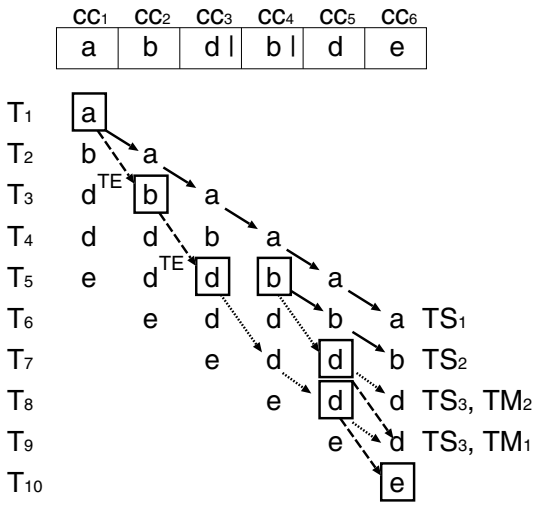


Fig. 4. Tentative matching.

shown in Fig. 4. Each cell can be classified into three types of cells. For each C-term $t_j$, the leftmost cell in $t_j$ is called a term start cell, the rightmost cell in $t_j$ is called a term end cell, and the remaining cells are called term middle cells. In Fig. 4, $CC_1$, $CC_4$, and $CC_5$ are term start cells, $CC_3$, $CC_4$, and $CC_6$ are term end cells, and $CC_2$ is a term middle cell. Note that, if a C-term consists of one symbol, the same cell is classified into both the term start and end cells. We also call all cells except the rightmost cell *general cells*. The rightmost cell is called a *decision cell*.

For any term start cell, after starting empty matching, the term start cell always starts tentative matching if an input text symbol is matched with a pattern symbol stored in this term start cell. In Fig. 4, assume that $T = abdde$, which is input from $CC_1$, and moves rightward one cell by one cell in each clock cycle. The cell $CC_1$ starts tentative matching in clock cycle $T_1$, $CC_4$ starts tentative matching in $T_5$, and $CC_5$ starts tentative matching in $T_7$ and $T_8$. When a term start cell starts tentative matching for C-term $t_j$, the tentative matching start signal $TS_j$ is set to *true*, and this signal is transferred in the systolic array one cell by one cell in each clock cycle. In Fig. 4, this signal is denoted as a solid line. The term start cell also sends the tentative matching enable signal $TE$ to its right neighbor cell in two clock cycles. In Fig. 4, this signal is denoted as a rough dotted line.

For any term middle cell, when it receives the tentative matching enable signal $TE$ from its left neighbor cell, it starts tentative matching, and if the matching is successfully done, then it sends the tentative matching enable signal $TE$ to its right neighbor cell in two clock cycles. If matching fails, the cell does not send $TE$ to its right neighbor cell.

For any term end cell receiving the tentative matching enable signal $TE$, it starts tentative matching, and if the matching is successfully done, then it sends the tentative matching success signal $TM_j$ to its right neighbor cell in two clock cycles. This tentative matching success signal $TM_j$ is transferred in the systolic array one cell by one cell in each clock cycle. In Fig. 4, this signal is denoted as a fine dotted line.

*3) Decision Cell:* As noted, string matching for UK-terms with the systolic algorithm is performed in two steps, tentative matching and final matching. In the following, we explain how final matching is performed.

As described, the rightmost cell is called a decision cell. In addition to functions of general cells, the decision cell performs final matching. As the case shown in Fig. 4, the cell $CC_6$ is a decision cell. The decision cell has a binary counter, called Text Counter, $TC$ and a set of First-In-First-Out (FIFO) buffers, called Term Matching Memories, $TMM_i$, $1 \leq i \leq t$, where $t$ is the number of C-terms in pattern $P$.

The initial value of $TC$ is set to 0, and it is incremented when the decision cell receives a text symbol from its left neighbor cell. Each FIFO buffer $TMM_i$ is set to empty when the algorithm starts.

When the decision cell receives the tentative matching start signal $TS_i$ from its left neighbor cell, and in the same clock cycle, the decision cell decides that empty matching or tentative matching for some C-term in $P$ is successfully done, then the current value of $TC+length_i$ is written into the FIFO buffer $TMM_i$, where $length_i$ is the length of $i$-th C-term. For example, in Fig. 4, in clock cycle $T_6$, the decision cell receives $TS_1$. In clock cycle $T_6$, as shown in Fig. 3, empty matching is successfully done. Thus, $TC + length_1 = 1 + 3 = 4$ is written into FIFO buffer $TMM_1$. In clock cycle $T_7$, the decision cell receives $TS_2$. However, in this clock cycle, no tentative matching is successfully done. Thus, no data will be written into FIFO $TMM_2$.

For each clock cycle, each FIFO buffer $TMM_i$ is checked. Let the oldest data stored in the FIFO buffer $TMM_i$ be $top_i$. If the value of $top_i$ is equal to the value of counter $TC$, and in the same clock cycle, the decision cell receives the tentative matching success signal $TM_i$, then the decision cell decides that tentative matching for C-term $t_i$ has been successfully done, and the matching enable signal $E_{out}$ is set to *true* by setting flag $E$ to *true*. Otherwise, the data $top_i$ is simply discarded from the buffer. For example, in Fig. 4, in clock cycle $T_8$, the decision cell receives $TM_2$. However, the FIFO buffer $TMM_2$ is empty. Thus, tentative matching has been failed. In clock cycle $T_9$, the decision cell receives $TM_1$. The FIFO buffer $TMM_1$ holds 4 as the oldest value. Since this is equal to the value of $TC$, tentative matching has been successfully done.

*4) Behavior of Cells:* The algorithm descriptions of general and decision cells are given in Fig. 5 and Fig. 6. As noted, the behavior of each cell is synchronized to a global clock signal, and we assume a two-phase clocking scheme, and each clock cycle $T$ consists of a pair of two clock signals $(t_0, t_1)$. Each cell repeats this procedure until a given text is output from the rightmost cell. In these descriptions, we omit the pattern input phase. 'MOVE_INPUT_STRING' means that a given text is shifted right by one cell. Each cell has a special register $TERM$, whose value is set to $i$, if the cell has a symbol in

```
t0: begin
      MOVE_INPUT_STRING;
      if OP='(' then E1:=E_in or ME;
        else if U_in then E1:=ES_in or ME;
        else E1:=TE_in;
      if OP='(' then begin
        ES:=E_in; ME:=E_in or ME end;
        else begin ES:=ES_in;
          if U_in then ME:=ES_in or ME end;
      TS_i:=TS_i_in; TM_i:=TM_i_in;
      if OP='|' then begin
        TM_i:=TM1; TM2:=false end;
        else begin E:=false; TM2:=TM1 end;
      TM1:=false;
    end;
;
t1: begin
      if E1 then begin
        R:=CMP(LP, LC)
        if R or (LP='?') then begin
          TM1:=true;
          if (OP='(') or U_in then
            TS_i:=true end;
      end
end;
```

Fig. 5. Behavior of general cells.

$i$-th C-term $t_i$ in pattern $P$.

For each cell, if its left neighbor cell has "|" in register $OP$, then its input $U_{in}$ is set to *true*.

When implementing comparison cells, any cell has a capability of performing both general and decision cells. Note that, basically, a set of functions supported by a general cell is a subset of functions of the decision cell. When setting a pattern, if a cell contains ")" in register $OP$, it is designated as a decision cell, otherwise, it is designated as a general cell.

In the algorithm description of the decision cell in Fig. 6, $TOP_TMM(i)$ is a function, whose return value is the oldest data stored in the FIFO buffer $TMM_i$. $ADD_TMM(i, x)$ is a procedure, which stores $x$ in $TMM_i$. $DELETE_TMM(i)$ is a procedure, which deletes the oldest data stored in $TMM_i$. Let the width of the binary counter $TC$ be $W$. Then, any addition to $TC$ in the algorithm is actually an addition with modulo $2^W$. $W$ is specified as follows. Let $L$ be the longest length of a C-term contained in UK-term. Then, $W = \lceil \log_2 L \rceil$.

### F. Matching of a C-term

If a given pattern $P$ is a C-term, it is easy to realize a matching algorithm using comparison cells described in the previous subsection with a slight modification. Only when the leftmost cell, i.e., the term start cell, receives the enable signal $E_{in}$ from the outside, tentative matching of a given C-term begins. If tentative matching is successfully done, then the rightmost cell, i.e., the decision cell, outputs $E_{out}$ by setting flag $E$ to *true*. The other functions of comparison cells and decision cells, such as empty matching or final matching, are not utilized. Due to the lack of space, the detailed description is omitted here.

```
t0: begin
      MOVE_INPUT_STRING;
      TC:=TC+1;
      if U_in then E1:=ES_in or ME;
        else E1:=TE_in;
      ES:=ES_in;
      E:=false;
      if U_in then ME:=ES_in or ME;
      for all j such that (ji) in parallel
        if (TC=TOP_TMM(j))
          then begin E:=TM_j_in; T:=TM_j_in;
            DELETE_TMM(j) end;
      if (TC=TOP_TMM(i))
        then begin E:=TM1; T:=TM1;
            DELETE_TMM(i) end;
      E:=ES_in or E;
      if ES_in or T then
        for all j in parallel
          if TS_j_in then ADD_TMM(j, TC+length_j)
      TM1:=false
    end;
;
t1: begin
      if E1 then begin
        R:=CMP(LP, LC)
        if R or (LP='?') then begin TM1:=true;
          if U_in then begin TS_i:=true;
            if ES_in or T then
              ADD_TMM(i, TC+length_i); end end;
      end
end;
```

Fig. 6. Behavior of the decision cell.

### G. Matching of a U-term

If a given pattern $P$ is a U-term, it is easy to realize a matching algorithm using comparison cells described in the previous subsection with a slight modification. Only when the leftmost cell, i.e., the term start cell, receives the enable signal $E_{in}$ from the outside, tentative matching of a given U-term begins. If tentative matching for some C-term $t_j$ in $P$ is successfully done, then the rightmost cell, i.e., the decision cell, outputs $E_{out}$ by setting flag $E$ to *true*. The other functions of comparison cells and decision cells, such as empty matching or final matching, are not utilized. Due to the lack of space, the detailed description is omitted here.

### H. Matching of an Admissible Pattern

If a given pattern $P$ is a concatenation of C-, U-, and UK-terms, for each term $T_j$, corresponding comparison cells, denoted $CC_{j_1}, CC_{j_2}, \ldots, CC_{j_p}$, perform string matching of $T_j$. The comparison cell $CC_{j_1}$ starts matching when it receives the valid enable signal $E_{in}$ from its left neighbor cell. If matching of $T_j$ has been successfully performed, then the cell $CC_{j_p}$ sets $E_{out}$ to *true*, which starts matching of term $T_{j+1}$.

## III. Approximate String Matching

In this section, we introduce the hardware algorithm for calculating the edit distance and its architecture.

### A. Related Work on Approximate String Matching Engine

As previous results related to our study, for the problem of calculating the edit distance, Yu, *et al.* have also proposed a hardware algorithm to be implemented on an FPGA chip [27]. However, in this algorithm, character symbols in a pattern were restricted to A, C, G, and T, since their algorithm was originally proposed for the analysis of DNA sequences. It would be very difficult to extend this algorithm for the general approximate string matching problem. As far as we investigated, no previous results have been known for hardware implementation of a string matching engine for an arbitrary set of character symbols. In this section, we introduce an approximate string matching engine which calculates the edit distance for an arbitrary set of character symbols.

### B. Edit Distance

Let $A$ be a finite string (or sequence) or character (or symbols). $A < i >$ is the $i$th character of string $A$; $A < i : j >$ is the $i$th through $j$th characters (inclusive) of $A$ if $i \leq j$. $|A|$ denotes the length of string $A$.

An *edit operation* is a pair $(a, b) \neq (\Lambda, \Lambda)$ of strings of length less than or equal to 1, and usually written $a \rightarrow b$, where $\Lambda$ denotes the null string. String $B$ results from the application of the operation $a \rightarrow b$ to string $A$, written $A \Rightarrow B$ *via* $a \rightarrow b$, if $A = \sigma a \tau$ and $B = \sigma b \tau$ for some strings $\sigma$ and $\tau$. We call $a \rightarrow b$ a *change* operation if $a \neq \Lambda$ and $b \neq \Lambda$; a *delete* operation if $b = \Lambda$; and an *insert* operation if $a = \Lambda$.

Let $\gamma$ be an arbitrary cost function which assigns to each edit operation $a \rightarrow b$ a nonnegative real number $\gamma(a \rightarrow b)$. Extend $\gamma$ to a sequence of edit operations $S = s_1, s_2, \ldots, s_m$ by letting $\gamma(S) = \sum_{i=1}^{m} \gamma(s_i)$. We now let the *edit distance* $\delta(A, B)$ from string $A$ to string $B$ be the minimum cost of all sequences of edit operations which transform $A$ into $B$ [24].

Given a pair of strings $A$ and $B$, the *approximate string matching problem* is to find the edit distance between two strings $A$ and $B$. For this problem, the following theorem holds [24].

*Theorem 1:* Let $A(i) = A < 1 : i >$ and $B(j) = B < 1 : j >$, and $D(i, j) = \delta(A(i), B(i))$, $0 \leq i \leq |A|$, $0 \leq j \leq |B|$. Then,

$$
\begin{aligned}
D(i,j) = \min\{ & \\
& D(i-1, j-1) + \gamma(A < i > \rightarrow B < j >), \\
& D(i-1, j) + \gamma(A < i > \rightarrow \Lambda), \\
& D(i, j-1) + \gamma(\Lambda \rightarrow B < j >)\} \qquad (1)
\end{aligned}
$$

for all $i$, $j$, $1 \leq i \leq |A|$, $1 \leq j \leq |B|$. □

From Theorem 1, for given two strings $A$ and $B$, the edit distance $\delta(A, B)$ from string $A$ to string $B$ is given as $D(|A|, |B|)$. In this section, the matrix $D$ is called the *edit distance matrix*.

In this section, we slightly extend the approximate string matching problem, and formulate this extended problem as the *multiple string matching problem*. Given a set of finite strings $R = \{S_1, S_2, \ldots, S_m\}$ and a *pattern* string $P$, the multiple string matching problem is to calculate the edit distance $\delta(P, S_i)$ from string $P$ to string $S_i$ for all $i$, $1 \leq i \leq m$. We call each string $S_i$ to be matched a *text string*. When $m = 1$, this problem is equivalent to the original approximate string matching problem. Applications of this problem include the text retrieval in large text database, and DNA sequence alignment in bioinformatics.

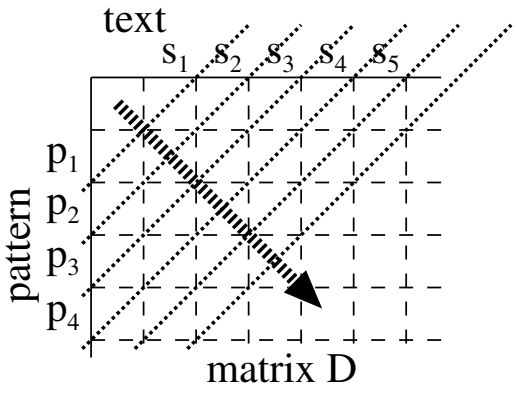


Fig. 7. Parallel calculation of the edit distance matrix.

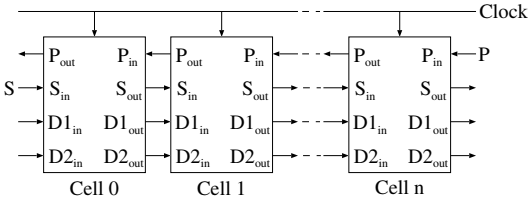


Fig. 8. Systolic architecture.

### C. Basic Algorithm

In this section, for the multiple string matching problem, we introduce a hardware algorithm, which is implemented on FPGAs, to realize a high-speed calculation of edit distance. The basic idea of the proposed algorithm is as follows. From Theorem 1, for a given pair of two strings, the edit distance is obtained by computing the edit distance matrix. In this matrix computation, one can easily understand that all entries on any positive slope diagonal lines can be computed in parallel, since there are no data dependencies among them. Fig. 7 shows how to compute the edit distance matrix in parallel. The basic idea of the proposed algorithm is to assign a processing element to each row of the edit distance matrix, and all processing units calculate the values of matrix elements on each positive slope diagonal line in parallel.

Fig. 8 shows the overview of the architecture. It consists of $(n+1)$ simple processing units, called *cells*, where $n = |P|$. As mentioned, the entire algorithm calculates diagonal elements in the edit distance matrix in the parallel and pipeline fashion. In the following, the details of the algorithm are explained.

*1) Inputs of the Algorithm:* For a given pattern string $P = p_1 p_2 \ldots p_n$, let $P' = \theta p_1 p_2 \ldots p_n$, and each character in $P'$ is stored in each cell in advance, where $\theta$ is a special start

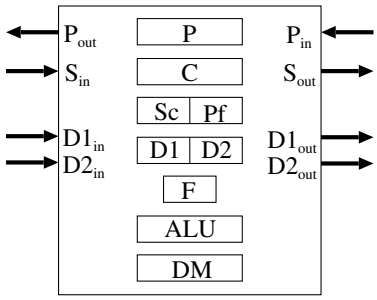


Fig. 9. The cell.

```
t0: begin
  C ← Sin;
  if Sin = θ then Sc ← true;
  if Sin = λ then Sc ← false;
  F ← (Sin = θ) ∨ (Sin = ",") ∨ (Sin = λ);
  sub ← DM(P, C);
end;
t1: begin
  if Sc then
    case (Pf, F) begin
      00: D1 ← min{D1in + del,
                   D1 + ins, D2in + sub};
      01: D1 ← D1in + del;
      10: D1 ← D1 + ins;
      11: D1 ← 0;
    end;
  D2 ← D1;
end;
```

Fig. 10. The algorithm of a cell.

symbol. The pattern string is input from the rightmost cell. On the other hand, a given set of text strings to be matched is concatenated and sequentially input from the leftmost cell. When concatenating multiple text strings, special symbol "," is inserted as a delimiter, and $\theta$ and $\lambda$ are added as the first and last symbols, respectively. For example, if $S_1 = abb$, $S_2 = cba$, and $S_3 = acb$ then the text string to be input is $S = \theta abb, cba, acb\lambda$.

*2) The Cell:* The structure of a cell is shown in Fig. 9. Each cell consists of two latches $P$ and $C$, which are used to store characters in $P$ and $S_i$. The cost function $\gamma$ is stored in the memory $DM$ in each cell. $DM$ is realized as a two-dimensional array of words, and is called the *edit cost matrix*. For any pair of two characters $a$ and $b$, $DM(a, b)$ returns the value of $\gamma(a \rightarrow b)$. The values of all elements of the edit cost matrix are set in advance before starting the string matching. $D1$ and $D2$ are also latches, which store values of the edit distance matrix $D$. $Sc$, $Pf$ and $F$ are flags to be used in the cell algorithm.

*3) Behavior of the Cell:* As noted, the behavior of each cell is classified into two phases, the pattern input phase and the text matching phase. In the former phase, the pattern string is input from the rightmost cell one character by one character, and shifted left until all characters are stored in corresponding cells. Any cell which stores the start symbol $\theta$ in latch $P$ sets

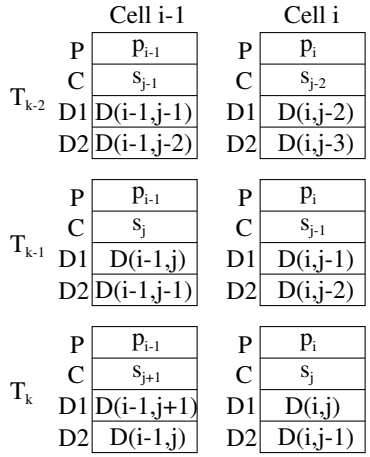


Fig. 11. Behavior of the algorithm.

the flag $Pf$ to 1, and any other cell sets $Pf$ to 0. In the text matching phase, the actual text matching is performed. In the following, we only describe the cell behavior of the text matching phase.

Fig. 10 shows the behavior of a cell during the text matching phase. We assume that each clock cycle consists of two clock phases $(t_0, t_1)$. We also assume that the delete and insert costs of any character are the same values, and denoted as $del$ and $ins$, respectively. For each clock cycle, each cell repeatedly executes this algorithm.

Cell $i$ stores pattern character $p_i$, and it calculates all elements $D(i, *)$ of the edit distance matrix. Assume that cell $i$ receives text string character $s_i$ from its left neighbor cell, stores it in latch $C$, and starts calculating $D(i, j)$ in clock cycle $T_k$. Fig. 11 shows this situation. From Theorem 1, to calculate $D(i, j)$, values of $D(i, j-1)$, $D(i-1, j)$, and $D(i-1, j-1)$ are required. Cell $i$ holds $D(i, j-1)$ in latch $D1$, which was calculated in clock cycle $T_{k-1}$. $D(i-1, j)$ was calculated also in clock cycle $T_{k-1}$ in cell $i-1$, and stored in $D1$. $D(i-1, j-1)$ was calculated in clock cycle $T_{k-2}$ in cell $i-1$, and stored in $D1$. This value was shifted to $D2$ in clock cycle $T_{k-1}$. Thus, all values required to calculate $D(i, j)$ are stored in cell $i$ or cell $i-1$.

When cell $i$ receives $\theta$ or "," or "$\lambda$", then the latch $D1$ is initialized, and set to 0. It means that matching for a next text string is started.

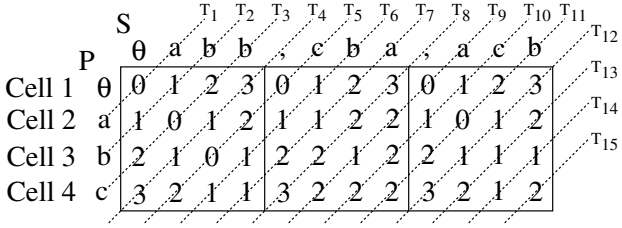


Fig. 12. Example of string matching.

Fig. 12 shows how string matching was processed by the proposed algorithm when $P = abc$ and $R = \{S_1, S_2, S_3\}$, where $S_1 = abb$, $S_2 = cba$, and $S_3 = acb$. From this figure, it is easy to understand that the proposed algorithm solves the multiple string matching problem in $O(L)$ clock cycles, where $L$ is the total length of input string $R$.

*4) Memory Structure:* As mentioned, each cell has its own copy of the edit cost matrix $DM$, in which matrix element $DM(a, b)$ represents the change cost from character $a$ to character $b$, i.e., $\gamma(a \rightarrow b)$. When implementing the proposed algorithm on FPGAs, the edit cost matrices are implemented by using the block RAMs. When the edit cost matrix is symmetric, it is easy to reduce the total memory size to its half size with a simple address transformation.

In the algorithm shown above, the insert and delete costs are assumed to be fixed. However, it is easy to extend the algorithm so that arbitrary insert and delete costs are allowed by extending the edit cost matrix. We newly introduce a special symbol, denoted $\Lambda$ to show the null character. Then, $DM(a, \Lambda)$ represents the delete cost of character $a$, and $DM(\Lambda, b)$ represents the insert cost of character $b$. Before starting string matching, each cell reads the delete cost from the edit cost matrix for a pattern character which the cell has in latch $P$, and stores it in a register. The leftmost cell reads the insert cost from the edit cost matrix for a text character which the cell receives from the input terminal, and stores it in another register. This value will be shifted right as the text string character is moved right. Note that the performance would be degraded if each cell reads the insert and delete costs from the edit cost matrix when $D(i, j)$ is calculated, since it would require three times of memory accesses.

## IV. FPGA Implementation Results

To evaluate the effectiveness of our hardware accelerators for string matching, we designed the accelerators with Verilog-HDL, and implemented them on an FPGA board, which has a Xilinx FPGA chip XC4VLX100-11F1513, using Xilinx ISE Version 8.2i as the FPGA design tool. The following subsections show the results of each accelerators.

### A. Results of Regular Expression Maching Engine

In Section II, we assume a 2-phase clock scheme. However, in the actual design of the circuit, we adopted a single phase clock scheme since for FPGA design, a single phase clock is normally used. The circuit has two states, denoted $T_0$ and $T_1$, and those are corresponding to $t_0$ and $t_1$ in the circuit behavior in Section II. Furthermore, those two states are executed in a pipeline fashion, i.e., $T_1$ of the circuit for matching an $i$-th symbol in the text is overlapped with $T_0$ for matching an $(i+1)$-th symbol in the text. Hence, pattern matching for one symbol in the text can be performed in one clock cycle.

In current design, one symbol in text and pattern consists of $8$ bits, and the maximum length of a pattern is $192$ due to the limit of the number of LUTs in the FPGA chip. The HDL description of the circuit consists of $8,150$ lines. As the result of logic synthesis, $91,830$ LUTs were utilized. The estimated maximum clock frequency reported by the design tool was $182$ MHz. Due to the restriction of the FPGA board, the actual clock frequency used in experiments was set to $125$ MHz.

In experiments, we prepared several patterns, and measured the execution time of string matching by the accelerator. For each case, we verified the result, and made sure that the correct result was produced. Since the accelerator can perform string matching in one clock per one symbol, the maximum throughput of the accelerator was approximately $182,000,000$ symbols per second, if the clock frequency was set to $182$ MHz. Since one symbol consists of $8$ bits, this was equivalent to $1.456$ Gigabits per second (Gps). On the FPGA board, a throughput of $1.0$ Gps was achieved when the clock frequency of the board was set to $125$ MHz.

TABLE I
Results of approximate string matching engine.

| Algorithm | #LUT | Clock [MHz] | Time [$\mu$s] | Ratio |
|---|---|---|---|---|
| Software | – | 3600 | 1,040,000 | 1 |
| Hardware | 12248 | 165 | 729 | 1427 |

### B. Results of Approximate String Maching Engine

We have also developed a software program for solving the multiple string matching problem, and compared it with our hardware algorithm implemented on the FPGA board. The software program was executed on a PC with a Pentium 4 3.6GHz CPU.

Table I shows the experimental results. In this table, "Software" shows the result of software program, and "Hardware" shows the result of the hardware algorithm described in Section III. "#LUT", "Clock", "Time" and "Ratio" show the number of LUTs used to implement the circuit, the clock frequencies of the CPU for software and the FPGA chip for hardware, the execution time, and the speedup ratio of the hardware algorithm compared to the software program, respectively. The length of a pattern was 120 for the cases of "Software" and "Hardware". The total length of input strings to be matched with a pattern was set to $120,000$.

From the experimental results, we see that the proposed hardware string matching engine drastically outperformed the software program.

We would also like to point out that, since the proposed hardware algorithm has an simple one-dimensional systolic architecture, it is easy to implement the algorithm for longer patterns by connecting multiple FPGA chips.

## V. Conclusion

We have introduced hardware algorithms for the two string matching problems, which are regular expression matching and the string-to-string correction problems, and shown their FPGA implementation results. Experimental results showed the effectiveness of the hardware algorithms. The future work on these research is as follows. First, extending a subclass

of regular expressions is interesting and important. Second, it is better to introduce convenient forms for describing regular expressions. The syntax adopted in the programming language Perl is a good example of convenient expressions of regular expressions [22]. Third, our regular expression matching engine was not area-efficient, that is, much hardware resource was required to implement it, compared with regular expression matching engine based on a pattern-specific approach (i.e., a pattern was embedded as hardware during the circuit design). Therefore, designing an area-efficient, as well as pattern-independent, regular expression matching engine will be a difficult, but, important, challenging, and interesting work in the future. Finally, we will try to develop hardware algorithms for different kinds of string matching. In particular, besides the string-to-string correction problem, there are many other problems, for which dynamic programming algorithms have been known in bioinformatics [8]. It is interesting and important to develop hardware algorithms for those problems.

## ACKNOWLEDGMENTS

This research was supported in part by Grant-in-Aid for Scientific Research (C)(No.20500054) from Japan Society for the Promotion of Science.

## REFERENCES

[1] J. Aoe, *Computer Algorithms: String Pattern Matching Strategies,* IEEE Computer Society Press, 1994.

[2] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," *Proc. 2006 IEEE International Conference on Field Programmable Technology,* pp.119–126, 2006.

[3] J. Divyasree, H. Rajashekar, and K. Varghese, "Dynamically reconfigurable regular expression matching architecture," *Proc. International Conference on Application-Specific Systems, Architectures and Processors (ASAP 2008),* pp.120–125, July 2008.

[4] M. J. Foster and H. T. Kung, "The design of special-purpose VLSI chips," *IEEE Computer,* Vol.13, No.1, pp.26–40, 1980.

[5] P. A. V. Hall and G. R. Dowling, "Approximate string matching," *ACM Computing Surveys,* Vol.12, No.4, pp.381–402, 1980.

[6] J. T. L. Ho and G. G. F. Lemieux, "PERG: A scalable FPGA-based pattern-matching engine with consolidated bloomier filters," *Proc. 2008 IEEE International Conference on Field Programmable Technology,* pp.73–80, Dec. 2008.

[7] J. E. Hopcroft, J. D. Ullman, and R. Motwani, *Introduction to Automata, Theory, Languages and Computation, Second Edition*, Addison-Wesley, 2000.

[8] N. C. Jones and P. A. Pevzner, *An Introduction to Bioinformatics Algorithms,* The MIT Press, 2004.

[9] Y. Kawanaka, S. Wakabayashi, and S. Nagayama, "Design and FPGA implementation of a high-speed string matching engine," *Proc. 14th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI'07),* pp.122–129, 2007.

[10] Y. Kawanaka, S. Wakabayashi, and S. Nagayama, "A systolic regular expression pattern matching engine and its application to network intrusion detection," *Proc. 2008 IEEE International Conference on Field Programmable Technology,* pp.297–300, 2008.

[11] Y. Kawanaka, S. Wakabayashi, and S. Nagayama, "A fast regular expression matching engine for an FPGA-based network intrusion detection system," *Proc. 15th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI'09),* pp.88–93, 2009.

[12] Y. Kawanaka, S. Wakabayashi, and S. Nagayama, "A systolic string matching algorithm for high-speed recognition of a restricted regular set," *Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'09),* pp.151–157, 2009.

[13] T. Kikuno, N. Yoshida, and S. Wakabayashi, "Hardware algorithms for computing longest common subsequence," *Trans. IECE*, Vol.J65-D, No.8, pp.997–1004, 1982, in Japanese.

[14] H. T. Kung, "Why systolic architectures?," *IEEE Computer*, Vol.15, No.1, pp.37–45, 1982.

[15] D. L. Lee and F. H. Lochovsky, "HYTREM — A hybrid text-retrieval machine for large databases," *IEEE Transactions on Computers*, Vol.39, No.1, pp.111–123, 1990.

[16] C. Mead and L. Conway, *Introduction to VLSI Systems,* Addison-Wesley, 1980.

[17] S. Mikami, Y. Kawanaka, S. Wakabayashi, and S. Nagayama, "Efficient FPGA-based hardware algorithms for approximate string matching," *Proc. 23rd International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC'08),* pp.201–204, 2008.

[18] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," *Proc. 2007 ACM/IEEE Symposium on Architecture for Networking and Communications Systems,* pp.127–136, Dec. 2007.

[19] A. Mukhopadhyay, "Hardware algorithms for nonnumeric computation," *IEEE Transactions on Computers*, Vol.C-28, No.6, pp.384–394, 1979.

[20] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings*, Cambridge University Press, 2002.

[21] H. C. RoanW. J. Hwangand C. T. Dan Lo, "Shift-or circuit for efficient network intrusion detection pattern matching," *Proc. International Conference on Field Programmable Logic and Applications,* pp.785–790, 2006

[22] R. L. Schwartz and T. Phoenix, *Learning Perl, 3rd Edition*, Oreilly & Associates Inc., 2001.

[23] Y. Sugawara, M. Inaba, and K. Hiraki, "Over 10Gbps string matching mechanism for multi-stream packet scanning systems," *Proc. International Conference on Field Programmable Logic and Applications,* pp.484–493, Aug. 2004

[24] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of ACM*, Vol.21, No.1, pp.168–173, 1974.

[25] S. Wakabayashi, "Pattern matching machines for recognizing restricted regular sets suitable for VLSI implementation," *IECE Technical Report*, AL85-33, 1985, in Japanese.

[26] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character NFA," *Proc. International Conference on Field Programmable Logic and Applications,* pp.131–136, Aug. 2008

[27] C. W. Yu, K. H. Kwong, K. H. Lee, and P. H. W. Leong, "A Smith-Waterman systolic cell," *Proc. FPL2003,* LNCS 2778, pp.375–384, 2003.

[28] S. Yusuf and W. Luk, "Bitwise optimised CAM for network intrusion detection systems," *Proc. International Conference on Field Programmable Logic and Applications,* pp.444–449, Aug. 2005