Processor Array Architectures for Deep Packet Classification

Fayez Gebali, *Senior Member*, *IEEE Computer Society*, and A.N.M. Ehtesham Rafiq

Abstract—This paper presents a systematic technique for expressing a string search algorithm as a regular iterative expression to explore all possible processor arrays for deep packet classification. The computation domain of the algorithm is obtained and three affine scheduling functions are presented. The technique allows some of the algorithm variables to be pipelined while others are broadcast over system-wide buses. Nine possible processor array structures are obtained and analyzed in terms of speed, area, power, and I/O timing requirements. Time complexities are derived analytically and through extensive numerical simulations. The proposed designs exhibit optimum speed and area complexities. The processor arrays are compared with previously derived processor arrays for the string matching problem.

Index Terms—Processor array, string search, deep packet classification, parallel hardware.

1 Introduction

The string matching problem is employed in packet classification, computational biology, spam blocking, and information retrieval, to mention only a few applications. String search operates on a given alphabet set Σ of size $|\Sigma|$, a pattern $P = p_0 p_1 \cdots p_{m-1}$ of length m, and a text string $T = t_0 t_1 \cdots t_{n-1}$ of length n, with $m \leq n$. The problem is to find all occurrences of pattern in the text string.

The average time complexity for implementing the string search problem on a single processor was proven to be $\mathcal{O}(n)$ [1]. To meet the requirement of fast string matching, several hardware solutions were proposed that made use of advances in Very Large Scale Integration (VLSI) and processor array design techniques. Processor arrays are simple, regular, and modular structures for implementing several recursive algorithms [2], [3], [4]. Several authors developed techniques for mapping regular iterative algorithms onto processor arrays [3], [4], [5], [6], [7], [8], [9]. This paper presents a systematic methodology for obtaining several processor array architectures for deep packet classification based on the techniques developed in [9].

Packet classification refers to the identification and classification of individual data packets arriving at a switch. There are three types of packet classification tasks [10]: 1) Single-field classification (SFC) looks at a single field in the packet header and is used mostly in packet routing. 2) Multifield classification (MFC) scans multiple fields of a packet header to classify packets and support quality of service (QoS) policies. 3) Deep packet classification (DPC) [10], [11] examines the packet payload data in order to make classification decisions for the high-level applications. This paper deals with a hardware support for the DPC.

The need for DPC is increasing rapidly with the emerging content-aware applications, such as content-switching, load balancing, data streaming, policy-based firewalls, intrusion detection, etc. For such applications, traditional look-up table and CAM (content-addressable memory)-based search engines are not suitable [11], [12]. A string search algorithm-based search engine is the most suitable for those applications [11], [13]. Several efficient linear string search algorithms have been developed [1], [14], [15]. Most of these algorithms use preprocessing to speed-up their search operations. This preprocessing requires search operations and data index update. These preprocessing operations do not use regular or iterative operations, thus making them unsuitable for processor array implementation. In [1], we proposed an algorithm that achieves better performance without any preprocessing. But, that algorithm is suitable for the single processor based hardware. In this paper, we deal with processor array-based hardware solutions.

A hardware implementation for the algorithmic search engine for packet classification can be assumed to have the following characteristics:

- The text length n is typically big and variable depending on the packet payload.
- The pattern length m varies from a word of few characters to hundreds of characters (e.g., a URL address).
- The word length w is determined by the data storage organization and datapath bus width.
- Typically, the search engine is looking for the existence of the pattern P in the text T, i.e., the search engine only locates the first occurrence of the P in T.
- The text string T is supplied to the hardware in word-serial format.

This paper is organized as follows: Section 2 discusses the literature related to parallel algorithms and hardwares for the string search problem. Section 3 introduces the systematic methodology we employed to design the processor array architecture. Sections 4, 5, and 6 describe the resulting processor arrays derived in Section 3. Section 7 discusses the complexity analyses of our proposed hardwares. We verify the analysis results of the time complexity

• *The authors are with the Department of Electrical and Computer Engineering, University of Victoria, Victoria BC, V8W 3P6, Canada. E-mail: {fayez, nrafiq}@engr.uvic.ca.*

Manuscript received 28 July 2004; revised 21 Mar. 2005; accepted 26 Apr. 2005; published online 25 Jan. 2006.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0186-0704.

1045-9219/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

in Section 8 by extensive numerical simulations. In Section 9, we compare our design with previously designed processor arrays for the string search algorithm. Finally, we conclude our paper in Section 10.

2 Related Works

Different researchers tried different approaches to speed up the string search problem using algorithmic and hardware techniques. In this section, we summarize their works under three categories.

2.1 Parallel String Search Algorithms

In this section, we discuss theoretical techniques for developing parallel string search algorithms.

Jájá has proposed a parallel algorithm for string searching in [16]. His proposed algorithm does several preprocessings before performing the actual search. It preprocesses T in $\mathcal{O}(\log_2 m)$ time using $\mathcal{O}(m)$ operations. It also preprocesses P in $\mathcal{O}(\log_2 m)$ time using $\mathcal{O}(m)$ operations. It does the actual searching in $\mathcal{O}(\log_2 m)$ time using $\mathcal{O}(n)$ operations. This technique is intended for programmable multiple processor systems. Processors do different tasks at different times.

In [17], [18], a constant-time randomized parallel string matching algorithm is proposed. These algorithms compute deterministic samples of a sufficiently long substring of the pattern. Some parameters are randomly chosen during implementation. These randomized algorithms require $\mathcal{O}(\log \log m)$ time for preprocessing and constant time for searching on a CRCW(concurrent-read concurrent-write) PRAM(parallel random-access machine). The PRAM is a shared-memory model of parallel computation which consists of a collection of identical processors and a shared memory. This complex technique is also intended for programmable multiple processor systems. Processors do different tasks at different times. In [19], Galil also has designed a CRCW-PRAM constant-time optimal parallel algorithm.

In [20], Misra uses the theory of powerlists [21] to develop a parallel string matching algorithm. It can search in $\mathcal{O}(\log n)$ time using $\mathcal{O}(nm)$ processors. This algorithm can even search wild card characters. The technique used in this paper helps to derive a parallel algorithm. The paper does not mention the type of the hardware that is suitable for its implementation.

In [22], Chung has proposed a string matching algorithm with variable length don't cares. The proposed algorithm can be performed in $\mathcal{O}(1)$ time on an $m \times n$ mesh-connected computer with a reconfigurable bus system using $\mathcal{O}(nm)$ processors. Bertossi and Logi also proposed an algorithm with variable length don't cares in [23]. But, their algorithm can search in $\mathcal{O}(\log n)$ time using $\mathcal{O}(mn/\log n)$ processors using the EREW (exclusive read exclusive write) PRAM.

2.2 Parallel Hardwares for String Searching

In this section, we summarize some hardwares (other than processor array) for the string search problem.

Takefuji et al., in [24], proposed an algorithm that requires $m(n-m+1)$ processing elements and $2m(n-m+1)$ comparators to search P after only two iterations. They have organized the processing elements into a neural network array. Although the algorithm's time requirement is good, the area requirement is very high.

Cheng and Fu [25] proposed the space-time domain expansion approach for the hardware implementation of string matching. The time complexity of their approach is $\mathcal{O}(n)$ using $m \times n$ processing elements. The algorithm's space-time complexity is high compared to other techniques. Also they use ad hoc implementation technique that needs verifications after implementation.

Isenman and Shasha [26] developed a hardware for string matching using a deterministic finite state automaton based on the standard technique of Knuth-Morris-Pratt algorithm [27]. The hardware consists of an AT&T 32100 microprocessor that implements the compiled code for the UNIX System command `fgrep`. The controller uses 28 single character comparators together with four 16 bit adders. The speed of the system depends on the complexity of the query, but the use of multiple comparators in parallel enables them to achieve performance of a factor of up to 500 compared to using no parallel preprocessing. They verified the effectiveness of their approach through extensive behavioral simulations.

2.3 Processor Array Designs for the String Search

In this section, we summarize some proposed processor array implementations for the string search problem.

Foster and Kung [28] indicated that the design of fast special purpose chips strongly depends on the correct choice of an underlying algorithm that has properties of modularity and regularity. These properties allow design of processor arrays using different design procedures. Thus, a good algorithm must have 1) few operations to be implemented using few simple cells, 2) local and regular data and control flow requirements, and 3) inherent pipelining and multiprocessing features. Regular Iterative Algorithms (RIAs) exhibit all these properties and the challenge is to identify such an algorithm for the problem at hand. The processor array, proposed by Foster and Kung, accepts two streams of characters from the host machine to represent the pattern and text. The output of the machine is a stream of bits each of which corresponds to one of the characters in the text string. We should note that such preassumptions about data arrivals and productions place constraints on possible processor arrays' hardware spaces. Foster and Kung identified a RIA suitable for the string matching problem and their assumptions about data arrivals forced them to use a hardware that is 50 percent efficient since only one-half of the cells are active at any clock cycle. They proposed alternate structures that eliminate this inefficiency. Perhaps another important contribution of their paper is identifying that classical algorithms such as Boyer-Moore are not suited for fast hardware implementations since they do not possess regularity or modularity.

Mukherjee [29] devised a processor array to compare two strings based on the longest common subsequence (LCS) technique in $\mathcal{O}(n+m)$ time. The processor arrays were based on dynamic programming and an iterative algorithm was developed for this problem. The proposed processor array had the text and pattern moving in opposite directions.

Park and George [30] developed a processor array using a data-flow technique. The run-time complexity of their approach is $\mathcal{O}(n/d + \alpha)$ using $d \times m$ processing elements, where α equals $\log m$ for parallel hierarchical scheme and m for parallel linear scheme, and d is the number of input streams. Their approach cannot use data parallelism efficiently. Parallelism is not applied when $d = 1$.

Michailidis and Margaritis developed a processor array for the string search problem in [31] that required preprocessing and search phases. The algorithm for the preprocessing phase was expressed as a regular iterative algorithm (RIA). The processor array for this phase was obtained using a data dependency graph and was mapped on the same processor array for the searching phase. The searching phase was implemented based on a data dependency graph for calculating a dynamic programming matrix. The dependence graph was transformed to a "local dependence graph" in order to ensure that input data is fed at the edge nodes. Data timing and projecting the graph nodes to processing elements (PEs) were done in one step.

In [32], Michailidis and Margaritis developed a processor array for the string search problem using nondeterministic finite automata. Like [31], they used dependency graph. Their approach has the same complexities and problems as in [31].

Sastry and Ranganathan [33] devised a processor array to calculate the edit distance between two strings based on dynamic programming. In their array, pattern is not searched in text. However, the approach can be applied in the string searching problem. The hardware requires $m + n - 1$ processing elements. The hardware has been designed and fabricated using 2-micron CMOS p-well technology. The time required to compare two strings is

$$\left(n + \left\lceil \frac{N}{2} \right\rceil\right) \times 25 \times 10^{-9} s, \tag{1}$$

where N is the number of processing elements. Equation (1) assumes that the processing can be completed in a single pass. If multiple passes are required, the required time is

$$\left((m-1) \times 2 \times \left\lceil \frac{N}{2} \right\rceil + n + \left\lceil \frac{N}{2} \right\rceil\right) \times 25 \times 10^{-9} s. \tag{2}$$

They did not give reasons for some of the design steps.

3 A Systematic Technique for Processor Array Design

Systematic techniques to design processor arrays allow for design space exploration for optimizing performance according to certain specifications while satisfying design constraints. Several techniques were proposed earlier [3], [4], [5], [9]. However, most of these techniques were only able to deal with two-dimensional (2D) algorithms such as one-dimensional digital filters design. They were all based on developing a data dependence graph (DG) as the starting point. Three-dimensional algorithms, such as matrix-matrix multiplication, could not be easily handled. A similar argument could be given for the case of designing two-dimensional filters for image processing since these algorithms give rise to four-dimensional data dependencies and it would be hard indeed to visualize or analyze the associated 4D dependence graph. The first author proposed a formal algebraic procedure for processor array implementation starting from a regular iterative algorithm with arbitrary dimensions [9]. The example given in that reference dealt with designing a processor array for a three-dimensional digital filter which gives rise to a dependence graph in a six-dimensional space. We develop here processor arrays for the string search problem using that formal technique. The steps we employ to design an optimized processor array for string matching are explained in the following sections.

```
Input T, P
for i = 0 to n − m do
    j ← 0
    while (j < m ⋀ t_{i+j} = p_j)
        j ← j + 1
    end while
    if (j = m)
        match_flag ← TRUE
        match_location ← i
        exit
    end if
end for
```

Fig. 1. The basic string search algorithm.

3.1 Expressing the Algorithm as an Iterative Expression

To develop a processor array, first we must be able to describe the string matching algorithm using recursions that convert the algorithm into a regular iterative algorithm (RIA). We can write the basic string search algorithm as in Fig. 1. This algorithm can also be expressed in the form of an iteration using two indices i and j.

$$y_i = \bigwedge_{j=0}^{m-1} \text{Match}\,(t_{i+j}, p_j), \qquad 0 \le i \le n-m, \tag{3}$$

where $y_i (Y, 0 \le i \le n-m)$ is a Boolean type output variable. If $y_i = \text{TRUE}$, then there is a match at position t_i, i.e., $t_{i:i+m-1} = p_{0:m-1}$. $Match(a, b)$ is a function that is true when character a matches character b. $\bigwedge$ represents an m-input AND function.

3.2 Obtaining the Algorithm Dependence Graph (DG)

The string matching algorithm of (3) is defined on a two-dimensional (2D) domain since there are two indices(i, j). Therefore, a data dependence graph can be easily drawn as shown in Fig. 2. The *computation domain* is the convex hull in the 2D space where the algorithm operations are defined as indicated by the grayed circles in the 2D plane [9]. The output variable Y is represented by vertical lines so that each vertical line corresponds to a particular instance of Y. For instance, the line described by the equation

$$i = 3 \tag{4}$$

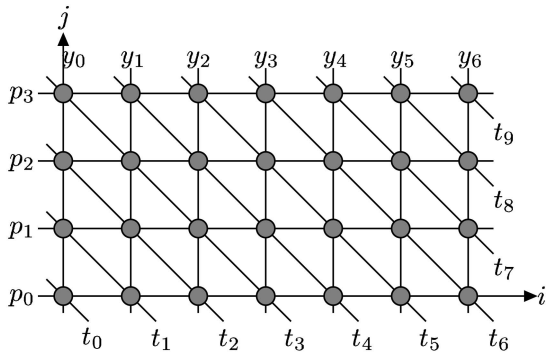

Fig. 2. Dependence graph for $m = 4$ and $n = 10$.

represents the output variable instance y_3. The input variable T is represented by the slanted lines. Again, as an example, the line represented by the equation

$$i + j = 3 \tag{5}$$

represents the input variable instance t_3. Similarly, the input variable P is represented by the horizontal lines.

3.3 Data Scheduling

Pipelining or broadcasting the variables of an algorithm is determined by the choice of a timing function that assigns a time value to each node in the DG. A simple but very useful timing function is an affine scheduling function of the form [9]

$$t(\mathbf{p}) = \mathbf{s}^t\mathbf{p} - s, \tag{6}$$

where the function $t(\mathbf{p})$ associates a time value t to a point $\mathbf{p}$ in the DG. The column vector $\mathbf{s} = [s_1\ s_2]$ is the *scheduling vector* and s is an integer.

A valid scheduling function uniquely maps any point $\mathbf{p}$ to a corresponding time index value. Such affine scheduling function must satisfy several conditions in order to be a valid scheduling function as explained below.

Input data timing restricts the space of valid scheduling functions. We assume the input text $T = t_0t_1 \cdots t_{n-1}$ arrives in word serial format where the index of each word corresponds to the time index. This implies that the time difference between adjacent words is one time step. Take the text instances at the bottom row nodes in Fig. 2 characterized by the line whose equation is $j = 0$. Two adjacent words, t_i and t_{i+1} at points $\mathbf{p}_1 = (i, 0)$ and $\mathbf{p}_2 = (i + 1, 0)$ arrive at the time index values i and $i + 1$, respectively. Applying our scheduling function in (6) to these two points, we get

$$t(\mathbf{p}_1) = js_1 - s \tag{7}$$

$$t(\mathbf{p}_2) = (j + 1)s_1 - s. \tag{8}$$

Since the time difference $t(\mathbf{p}_2) - t(\mathbf{p}_1) = 1$, we must have $s_1 = 1$. Therefore, a scheduling vector that satisfies input data timing must be specified as

$$\mathbf{s} = [1 \quad s_2]^t. \tag{9}$$

This leaves two unknowns in the possible timing functions, mainly the component s_1 and the integer s.

If we decide to pipeline a certain variable whose null-vector is $\mathbf{e}$, we must satisfy the following inequality [9]

$$\mathbf{s}^t\mathbf{e} \neq 0. \tag{10}$$

We have only one output variable Y whose null-vector is $\mathbf{e}_Y = [0\ \ 1]$. If we want to pipeline Y, then the simplest valid scheduling vectors are described by

$$\mathbf{s}_1 = [1\ \ 1] \tag{11}$$

$$\mathbf{s}_2 = [1\ \ -1]. \tag{12}$$

On the other hand, to broadcast a variable whose null-vector is $\mathbf{e}$, we must have [9]

$$\mathbf{s}^t\mathbf{e} = 0. \tag{13}$$

If we want to broadcast Y, then from (13) and (9), we must have

$$\mathbf{s}_3 = [1\ \ 0]. \tag{14}$$

Broadcasting an output variable simply implies that all computations involved in computing an instance of Y must be done in the same time step.

Another restriction on system timing is imposed by our choice of the *projection operator* as explained in the next section.

3.4 DG Node Projection

The projection operation is a many-to-one function that maps several nodes of the DG onto a single node. Thus, several operations in the DG are mapped to a single processing element (PE). The projection operation allows for hardware economy by multiplexing several operations in the DG on a single PE. El-Guibaly and Tawfik [9] explained how to perform the projection operation using a *projection matrix* $\mathbf{P}$. To obtain the projection matrix we require to define a desired *projection direction* $\mathbf{d}$. The vector $\mathbf{d}$ belongs to the null space of $\mathbf{P}$. Since we are dealing with a two-dimensional DG, matrix $\mathbf{P}$ is a row vector and $\mathbf{d}$ is a column vector.

A valid projection direction $\mathbf{d}$ must satisfy the inequality [9]

$$\mathbf{s}^t\mathbf{d} \neq 0. \tag{15}$$

In the following three sections, we will discuss design space explorations for the three values of $\mathbf{s}$ obtained in (11)-(14).

4 Design 1: Design Space Exploration when $\mathbf{s} = [1\ \ 1]^t$

The feeding point of t_0 is easily determined from Fig. 2 to be $\mathbf{p} = [0\ \ 0]^t$. The time value associated with this point is $t(\mathbf{p}) = 0$. Using (6), we get $s = 0$.

To study the timing of two input variables P and T, we first find their null-vectors:

$$\mathbf{e}_P = [1\ \ 0]^t \tag{16}$$

$$\mathbf{e}_T = [-1\ \ 1]^t. \tag{17}$$

The product of $\mathbf{s}$ and these two null-vectors gives

$$[1\ \ 1]\mathbf{e}_P = 1 \tag{18}$$

$$[1\ \ 1]\mathbf{e}_T = 0. \tag{19}$$

This choice for the timing function implies that input variable P will be pipelined and input variable T will be broadcast.

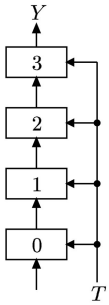

Fig. 3. Processor array for Design 1.a when $\mathbf{s} = [1 \;\; 1]^t$, $\mathbf{d}_a = [1 \;\; 0]^t$, and $m = 4$.

There are three simple projection vectors such that all of them satisfy (15) for the scheduling function in (11). The three projection vectors will produce three designs

$$\text{Design 1.a}: \;\; \mathbf{d}_a = [1 \quad 0]^t \tag{20}$$

$$\text{Design 1.b}: \;\; \mathbf{d}_b = [0 \quad 1]^t \tag{21}$$

$$\text{Design 1.c}: \;\; \mathbf{d}_c = [1 \quad 1]^t. \tag{22}$$

The corresponding projection matrices could be given by

$$\mathbf{P}_a = [0 \quad 1]^t \tag{23}$$

$$\mathbf{P}_b = [1 \quad 0]^t \tag{24}$$

$$\mathbf{P}_c = [1 \quad -1]^t. \tag{25}$$

Our processor design space now allows for three processor array configurations for each projection vector for the chosen timing function. In the following sections, we study the processor arrays associated with each design option.

4.1 Design 1.a: Using $\mathbf{s} = [1 \;\; 1]^t$ and $\mathbf{d}_a = [1 \quad 0]^t$

A point in the DG given by the coordinates $p = [i \quad j]^t$ will be mapped by the projection matrix $\mathbf{P}_a$ into the point

$$\mathbf{p}' = \mathbf{P}_a \mathbf{p} = j. \tag{26}$$

The processor array corresponding to Design 1.a is shown in Fig. 3. Input T is broadcast to all processors and word p_j of the pattern P is allocated to PE_j. The intermediate output of each PE is pipelined to the next PE with a higher index, as shown, such that the output samples y_i are obtained from the top PE. The processor array consists of m PEs and each PE is active for n time steps.

The PE details are shown in Fig. 4, where "D" denotes a 1-bit register to store the output.

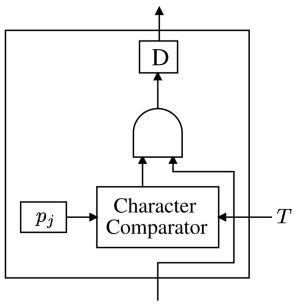

Fig. 4. PE detail for Design 1.a in Fig. 3.

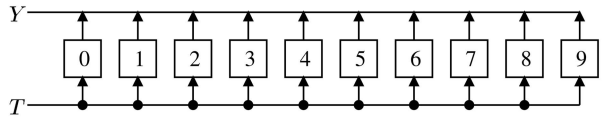

Fig. 5. Processor array for Design 1.b when $\mathbf{s} = [1 \; 1]^t$, $\mathbf{d}_b = [0 \;\; 1]^t$, and $n = 10$.

4.2 Design 1.b: Using $\mathbf{s} = [1 \;\; 1]^t$ and $\mathbf{d}_b = [0 \quad 1]^t$

A point in the DG given by the coordinates $\mathbf{p} = [i \quad j]^t$ will be mapped by the projection matrix $\mathbf{P}_b$ into the point

$$\mathbf{p}' = \mathbf{P}_b \mathbf{p} = i. \tag{27}$$

The resulting processor array is shown in Fig. 5.

The processor array consists of $n - m + 1$ PEs. Word p_i of the pattern P is fed to PE_0 and from there they are pipelined to the other PEs. The text words t_i are broadcast on the input bus to all PEs. Output y_i is obtained from PE_i at time i and a tristate buffer at the output of that PE ensures that it is the only output fed to the output bus. Each PE is active for m time steps only. Thus, the PEs are not well utilized as in the design of Section 4.1. However, we note from the DG of Fig. 2 that PE_0 is active for the time period 0 to $m - 1$ and PE_m is active for the time period m to $2m - 1$. Thus, these two PEs could be mapped to a single PE without causing any timing conflicts. In fact, all PEs whose index is expressed as

$$i' = i \bmod m \tag{28}$$

can all be mapped to the same processor without any timing conflicts. The resulting processor array after applying the above *modulo* operations on the array in Fig. 5 is shown in Fig. 6.

The processor array now consists of m PEs. The pattern P could be chosen to be stored in each PE or it could circulate among the PEs where initially PE_i stores the pattern word p_i. We prefer the former option since memory is cheap, while communications between PEs will always be expensive in terms of area, power, and delay. The text words t_i are broadcast on the input bus to all PEs. PE_i produces outputs i, $i + m$, $i + 2m$, $\cdots$ at times i, $i + m$, $i + 2m$, etc. The PE details are shown in Fig. 7. A tristate buffer at the output of that PE ensures that it is the only output fed to the output bus. The D register stores the output.

4.3 Design 1.c: Using $\mathbf{s} = [1 \;\; 1]^t$ and $\mathbf{d}_c = [1 \quad 1]^t$

A point in the DG given by the coordinates $p = [i \quad j]^t$ will be mapped by the projection matrix $\mathbf{P}_c$ into the point

$$\mathbf{p}' = \mathbf{P}_c \mathbf{p} = i - j. \tag{29}$$

The resulting processor array is shown in Fig. 8 for the case when $n = 10$ and $m = 4$, after adding a fixed increment to all PE indices to ensure non-negative PE index values.

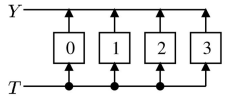

Fig. 6. Processor array for Design 1.b after applying the modulo operation in (28) for the case when $m = 4$.

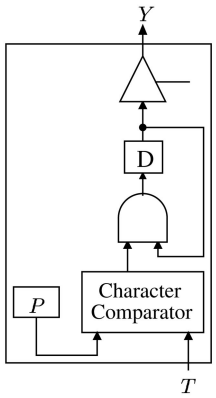

Fig. 7. Processing element for Design 1.b in Fig. 6.

The processor array consists of n PEs where only m of the processors are active at a given time step as shown in Fig. 9. At time step i, input text t_i is broadcast to all PE in the array.

We notice from Fig. 9 that at any time step only m out of the n processors are active. To improve PE utilization, we need to reduce the number of processors. An obvious processor allocation scheme could be derived from Fig. 9. In that scheme, operations involving the pattern word p_i are allocated to processor i. In that case, the processor array in Fig. 3 will result.

4.4 Comparing Designs 1.a and 1.b

Design 1.a in Section 4.1 performs better than Design 1.b in Section 4.2 for the following reasons:

- PE_j of Design 1.a (shown in Fig. 4) stores a single word of P (i.e., p_j) that can be stored in a register in the ALU. On the other hand, each PE in Design 1.b (shown in Fig. 7) stores the entire pattern P using on-chip memory module with its associated memory access delay [34].
- The clock period of Design 1.a (Fig. 3) is given by

$$\tau_{\text{clk}}(1.a) = \max[\tau_p + \tau_d, \tau_b], \tag{30}$$

where τ_p is the processing delay, τ_d is output driver delay, and τ_b is the input bus delay. τ_d is given by

$$\tau_d = \tau_0 \frac{C_l}{C_g}, \tag{31}$$

where τ_0 is the propagation delay when the output driver is loaded by a minimum-area inverter, C_l is the actual load capacitance, and C_g is the gate capacitance of a minimum-area inverter [35], [36].

The input bus delay τ_b is given by [37]

$$\tau_b = RC \times \frac{m(m+1)}{2} \approx 0.5\, RC\, m^2, \tag{32}$$

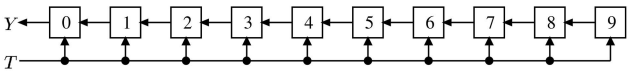

Fig. 8. Processor array for Design 1.c when $\mathbf{s} = [1\ \ 1]^t$, $\mathbf{d}_c = [1\ \ 1]^t$, $m = 4$, and $n = 10$.

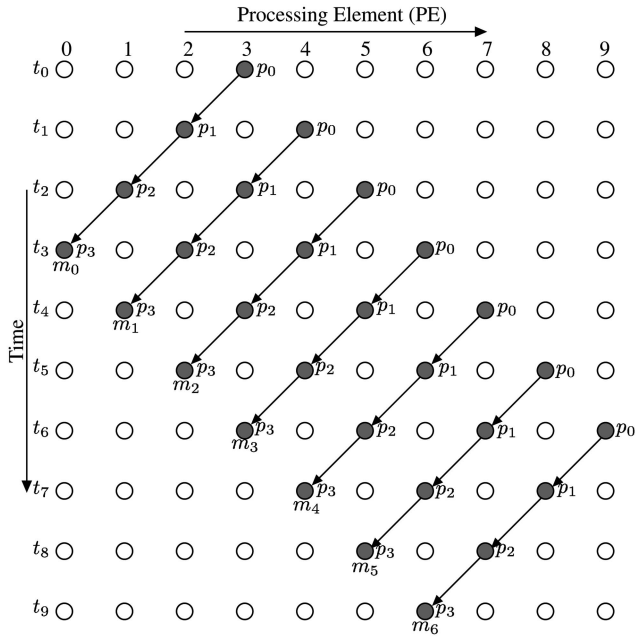

Fig. 9. Processor activity at the different time steps for the design in Fig. 8.

where R and C are the parasitic resistance and capacitance of one section of the bus between two adjacent PEs, respectively, and m is the number of PEs. Typically, τ_d is smaller than τ_b and, therefore, the clock period of Design 1.a equals τ_b (assuming $\tau_p \ll \tau_d$).

The clock period of Design 1.b (Fig. 5) is given by

$$\tau_{\text{clk}}(1.b) = \tau_p + \tau_b + \tau_m, \tag{33}$$

where τ_m is the memory access delay.

Comparing (30) and (33), we conclude that Design 1.a has slightly higher clock speed than Design 1.b.

- The area of each PE in Design 1.b is more than that of Design 1.a mainly due to the on-chip memory of size m.
- Power consumption in Design 1.a is given by

$$\varrho(1.a) = m\varrho_{PE} + \varrho_b, \tag{34}$$

where ϱ_{PE} is power consumed by each PE and ϱ_b is power consumed by the input bus.

Similarly, Power consumption in Design 1.b is given by

$$\varrho(1.b) = m\varrho_{PE} + 2\varrho_b. \tag{35}$$

Comparing (34) and (35), we conclude that Design 1.a consumes less power than Design 1.b.

In summary, Design 1.a is the best among the three designs from the point of view of speed, area, and power.

5 Design 2: Design Space Exploration when $\mathbf{s} = [1\ \ -1]^t$

Applying the scheduling function in (12) to $\mathbf{e}_P$ and $\mathbf{e}_T$, we get

$$[1\ \ -1]^t \mathbf{e}_P = 1 \tag{36}$$

$$[1\ \ -1]\mathbf{e}_T = 2. \tag{37}$$

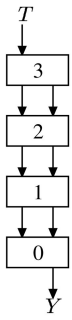

Fig. 10. Processor array for Design 2.a when $\mathbf{s} = [1 \ -1]$, $\mathbf{d}_a = [1 \ 0]^t$, and $m = 4$.

This choice for the timing function implies that both input variables P and T will be pipelined.

The pipeline direction for the input T flows in a south-east direction in Fig. 2. The pipeline for T is initialized from the top row in the figure defined by the line $j = m - 1$. Thus, the feeding point of t_0 is located at the point $\mathbf{p} = [-m \ \ m]^t$. The time value associated with this point is given by

$$t(\mathbf{p}) = -2m - s = 0. \tag{38}$$

Thus, the scalar s should be $s = -2m$. The processor arrays derived in this section will have a latency of $2m$ time units compared to Design 1.a given in Section 4.1.

There are three simple projection vectors such that all of them satisfy (15) for the scheduling function in Section 4.1. The three projection vectors are

$$\text{Design 2.a} : \mathbf{d}_a = [1 \quad 0]^t \tag{39}$$

$$\text{Design 2.b} : \mathbf{d}_b = [0 \quad 1]^t \tag{40}$$

$$\text{Design 2.c} : \mathbf{d}_c = [1 \quad -1]^t. \tag{41}$$

Our processor design space now allows for three processor array configurations for each projection vector for the chosen timing function. In the following sections, we study the processor arrays associated with each design option.

5.1 Design 2.a: Using $\mathbf{s} = [1 \ -1]^t$ and $\mathbf{d}_a = [1 \quad 0]^t$

Using the same treatment as in Section 4.1, the resulting processor array is shown in Fig. 10 for the case when $n = 10$ and $m = 4$. PEs of this design are same as shown in Fig. 4.

5.2 Design 2.b: Using $\mathbf{s} = [1 \ -1]^t$ and $\mathbf{d}_b = [0 \quad 1]^t$

Using the same treatment as in Section 4.2, the resulting processor array is shown in Fig. 11 for the case when $n = 10$ and $m = 4$. PEs of this design are same as shown in Fig. 7.

5.3 Design 2.c: Using $\mathbf{s} = [1 \ -1]^t$ and $\mathbf{d}_c = [1 \quad -1]^t$

The resulting processor array is similar to Design 1.c which, in turn, similar to Design 1.a in Section 4.1.

5.4 Comparing Designs 2.a and 2.b

All three designs, derived in the previous three subsections, have a latency of $2m$ clock periods before the first result appears. However, Design 2.a is better than Design 2.b for the following reasons:

- Design 2.a requires the least area since it does not require on-chip memory to store the pattern P.

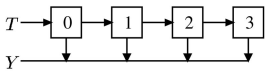

Fig. 11. Processor array for Design 2.b when $\mathbf{s} = [1 \ -1]^t$, $\mathbf{d}_a = [0 \ 1]^t$, and $m = 4$.

- The clock periods of Design 2.a and Design 2.b are given by

$$\tau_{\text{clk}}(2.a) = \tau_p + \tau_d \tag{42}$$

$$\tau_{\text{clk}}(2.b) = \tau_p + \tau_b + \tau_m. \tag{43}$$

  Since typically $\tau_d < \tau_b$, Design 2.a has higher clock speed than Design 2.b.
- Power consumptions of Design 2.a and Design 2.b are given by

$$\varrho(2.a) = m\varrho_{PE} \tag{44}$$

$$\varrho(2.b) = m\varrho_{PE} + \varrho_b. \tag{45}$$

  Thus, Design 2.a consumes less power than Design 2.b.

5.5 Comparing Designs 1.a and 2.a

Comparing (30) and (42), we conclude that Design 2.a is faster than Design 1.a. Comparing (34) and (44), we conclude that Design 2.a consumes less power than Design 1.a. Thus, so far, Design 2.a is the best design among the six designs proposed so far.

6 Design 3: Design Space Exploration when $\mathbf{s} = [1 \ 0]^t$

The feeding point of t_0 is easily determined from Fig. 2 to be $\mathbf{p} = [-m \ \ m]^t$. Time value of this point is $t(\mathbf{p}) = 0$. Using (6), we get $s = -m$. Thus, the processor arrays derived in this section will have a latency of m time units compared to Design 1.a given in Section 4.1.

Applying the scheduling function in (14) to $\mathbf{e}_P$ and $\mathbf{e}_T$, we get

$$[1 \ 0]\mathbf{e}_P = 0 \tag{46}$$

$$[1 \ 0]\mathbf{e}_T = -1. \tag{47}$$

This choice for the timing function implies that input variables P will be broadcast and T will be pipelined.

There are three simple projection vectors such that all of them satisfy (15) for the scheduling functions in (14). These projection vectors are

$$\text{Design 3.a} : \mathbf{d}_a = [1 \quad 0]^t \tag{48}$$

$$\text{Design 3.b} : \mathbf{d}_b = [1 \quad 1]^t \tag{49}$$

$$\text{Design 3.c} : \mathbf{d}_c = [1 \quad -1]^t. \tag{50}$$

Our processor design space now allows for three processor array configurations for each projection vector for the chosen timing function. In the following sections, we study the processor arrays associated with each design option.

6.1 Design 3.a: Using $\mathbf{s} = [1 \ 0]^t$ and $\mathbf{d}_a = [1 \quad 0]^t$

The processor array corresponding to Design 3.a is drawn in Fig. 12. PE_j stores only the value p_j, which can be stored in a register in the ALU similar to Design 1.a. The outputs of

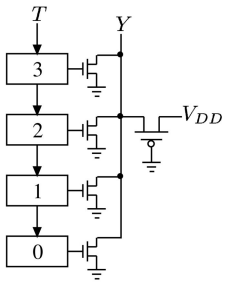

Fig. 12. Processor array for Design 3.a when $\mathbf{s} = [1\ \ 0]^t$, $\mathbf{d}_a = [1\ \ 0]^t$, and $m = 4$.

all PEs are wire-ORed or connected to the inputs of an m-input dynamic or static NOR gate as shown. This is the most efficient implementation that is also practical from the point of view of CMOS VLSI circuit considerations. The output of the NOR gate is in reality a long system-wide bus. As such, operating speed would suffer the same constraints that were discussed in Section 4.4. The processor array for Design 3.a is similar to Design 1.a, but all PEs operate on one output value at the same time.

6.2 Designs 3.b and 3.c: Using $\mathbf{s} = [1\ \ 0]^t$ and $\mathbf{d}_b = [1 \quad \pm 1]^t$

These two projection vectors produce the same processor array as Design 3.a. But, unlike Design 3.a, each PE stores the entire pattern P in the on-chip memory.

6.3 Comparing Designs 3.a, 3.b, and 3.c

Design 3.a in Section 6.1 performs the best among the three designs for the following reasons:

- Design 3.a requires the least area since it does not require on-chip memory to store the pattern P.
- All three designs are limited in speed by the delay of an m-input NOR gate. Although the outputs in all three designs are obtained through an m-input NOR gate, the gate speed is actually determined by the bus propagation delay. That bus is the output line connecting the driver transistors of the NOR gate. So, the clock periods of Design 3.a, Design 3.b, and Design 3.c are given by

$$\tau_{\text{clk}}(3.a) = \tau_p + \tau_{NOR} \approx \tau_p + \tau_b \tag{51}$$

$$\tau_{\text{clk}}(3.b) = \tau_p + \tau_b + \tau_m \tag{52}$$

$$\tau_{\text{clk}}(3.c) = \tau_p + \tau_b + \tau_m. \tag{53}$$

  Thus, Design 3.a has slightly higher clock speed than Design 3.b and Design 3.c.
- Power consumptions of all three designs are same and are given by

$$\varrho(3) = m\varrho_{PE} + \varrho_b. \tag{54}$$

6.4 Comparing Designs 3.a and 2.a

Comparing (42) and (51), we conclude that Design 2.a is faster than Design 3.a. Comparing (44) and (54), we conclude that Design 2.a consumes less power than Design 3.a. Thus, Design 2.a is the best design among the nine designs proposed in this paper.

7 Time Complexity Analysis

We provide, in this section, analyses of best, worst, and average times required to find a match. The time complexities reported have to be added with m for Designs 2. These time complexities also have to be scaled by the actual delay of one time step which depends on the particular design. For example, the time step associated with Designs 1 or Designs 2 is determined by the propagation delay of the output driver loaded by the adjacent PE. On the other hand, the time step associated with Designs 3 is determined by bus propagation delay that increases quadratically with the number of PEs.

7.1 Best Case

In the best case, y_0 will indicate a match. This output is obtained after m time steps.

7.2 Worst Case

In the worst case, all y_i outputs with $0 \le i < n - m$ will produce a negative result. Only the last output at position y_{n-m} produces a match. This output is obtained after n time steps.

7.3 Average Case

Assume a character of T matches a character of P with probability a. Assuming all characters are equally likely, a is given by

$$a = \frac{1}{|\Sigma|} = \frac{1}{2^w}, \tag{55}$$

where w is the number of bits in a character.

Define α_i as the probability of finding the first match at output y_i. In that sense, all outputs y_j with $0 \le j < i$ produced negative results. We can express α_i as

$$\alpha_i = a^m(1 - a^m)^{i-1}. \tag{56}$$

The average number of time steps for first match is given by

$$T_{av} = \sum_{i=0}^{n-m} (m + i)\ \alpha_i. \tag{57}$$

After a rather laborious algebraic manipulation (see the Appendix), we obtain

$$T_{av} = n - m. \tag{58}$$

8 Numerical Analysis

In this section, we perform extensive numerical simulations to estimate time complexities using the C programming language. The results of the numerical simulations are compared with the analytical results of Section 7.

We perform the simulations based on the following assumptions:

- Number of simulations = 100,000.
- $w = 32$ for typical 32-bit machine.
- Maximum value of n is 16,384 (which corresponds to the maximum network packet size).
- Maximum value of m is 25.
- P, T, m, and n are randomly generated. We use uniform distribution so that each value is equally likely.

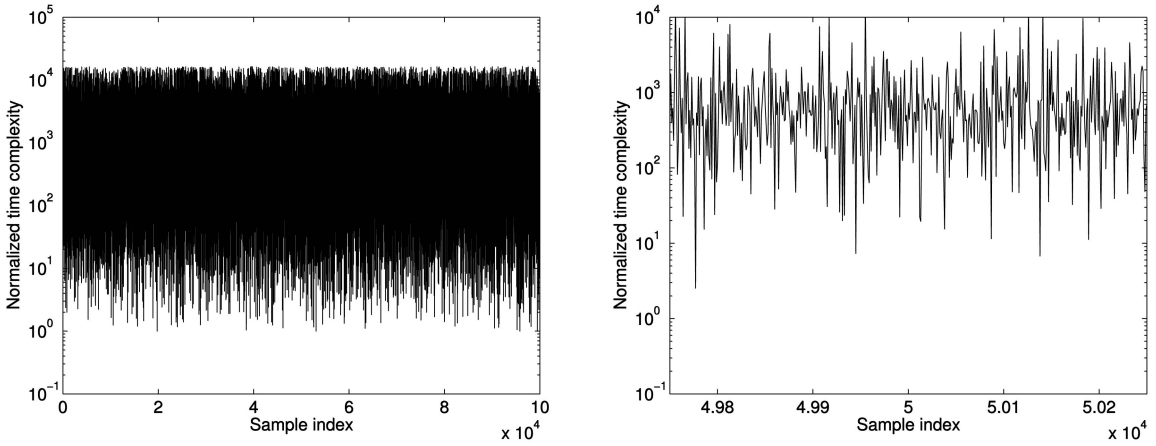

Fig. 13. Experimental results are plotted in (a) for all simulations and in (b) for 500 simulations. Results are normalized by m.

8.1 Best Case

The best case time complexity derived in Section 7 is m. Since m has been varied randomly in each simulation, we normalize each result by the corresponding m. Fig. 13a shows the graph of the normalized value vs. sample index. Fig. 13b shows the normalized results for 500 simulations taken from the middle of Fig. 13a. This allows us to see the fine scale variations. In Fig. 13, the minimum normalized value is 1. Thus, the best case time complexity is m as analytically derived in Section 7.

8.2 Worst Case

The worst case time complexity derived in Section 7 is n. Since n has been varied randomly in each simulation, we normalize each result by the corresponding n. Fig. 14a shows the graph of the normalized value vs. sample index. Fig. 14b shows the normalized results for 500 simulations taken from the middle of Fig. 14a. This allows us to see the fine scale variations. In Fig. 14, the maximum normalized value is 1. Thus, the worst case time complexity is n as derived in Section 7.

8.3 Average Case

The average case time complexity derived in Section 7 is $n - m$. Like Sections 8.1 and 8.2, we normalize each result by the corresponding $n - m$. Fig. 15a shows the graph of the normalized value versus sample index. We notice from this figure, for all simulations, the normalized search time is very close to 1 indicating that average search time is $n - m$ as was derived in Section 7. Fig. 15b is the histogram of the results of Fig. 15a. This figure shows that almost all the normalized results lie in the range between 0 and 3. So, we redraw the histogram in Fig. 15c in the range between 0 and 3. In Fig. 15, the average normalized value is 1. The median of the normalized results is also 1.0015. Thus, the average case time complexity is $n - m$ as derived in Section 7.

9 Design Comparison

In this section, we compare the technique we used to design the processor arrays with earlier techniques and we compare the designs we obtained with previously proposed processor arrays for the string search algorithm.

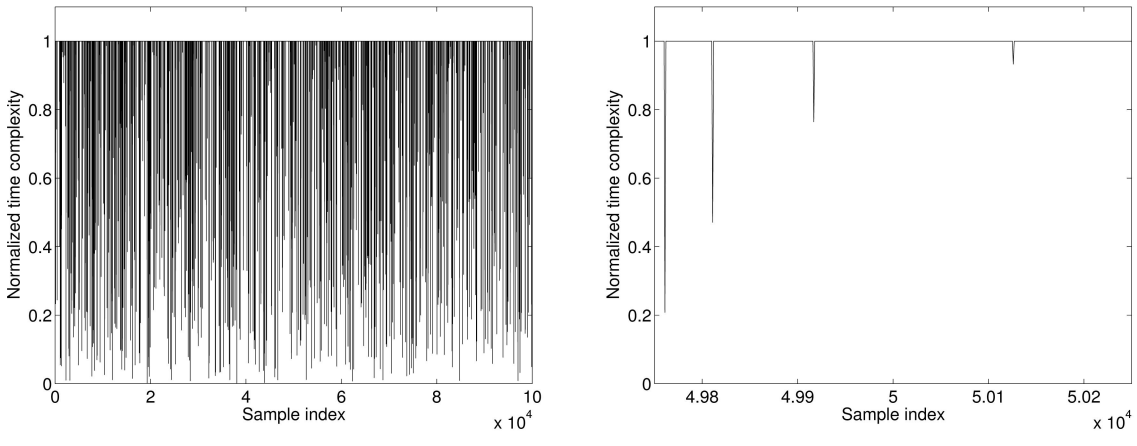

Fig. 14. Experimental results are plotted in (a) for all simulations and in (b) for 500 simulations. Results are normalized by n.

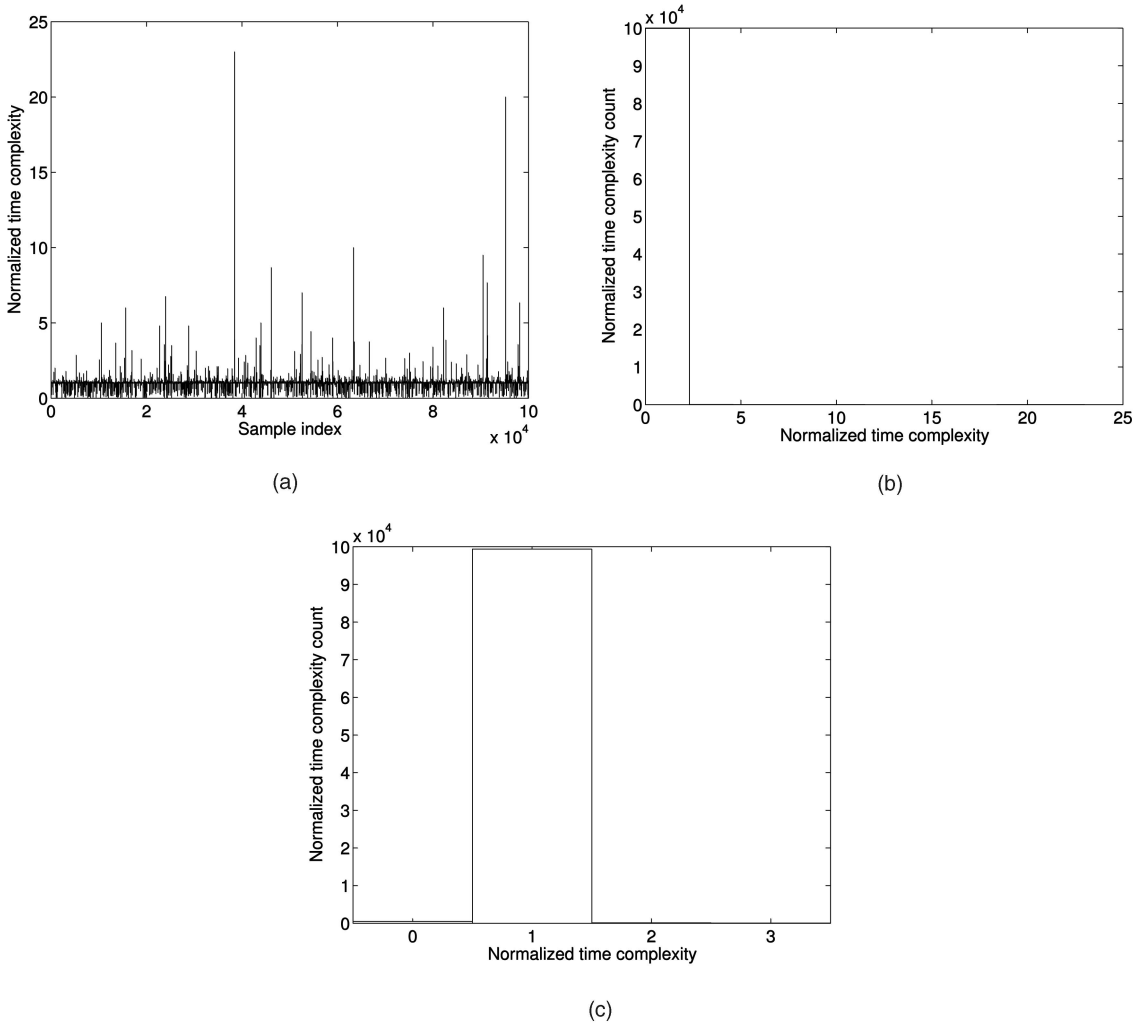

Fig. 15. (a) Experimental results are plotted in. (b) is the histogram of the results of (a). (c) is the same histogram in the range from 0 to 3. Results are normalized by $n - m$.

We employed a systematic technique to obtain our processor arrays by first converting the string search algorithm to a regular iterative expression (RIA). Having obtained the RIA, we were able to develop a data dependency graph (DG) which allowed us to explore possible data timing options that conform to I/O timing requirements. Earlier approaches did not explain how the designs were obtained or ad hoc techniques were used. Such techniques at best help develop one design and do not allow for design space exploration.

Design 2.a in Section 5.1 is identical to the one obtained by Foster and Kung [28]using ad-hoc techniques. Design 2.a is also similar to that proposed by Park and George [30]. Similar processor array has also been derived by Sastry et al. in [33].

The processor array of Mukherjee [29] determines the similarity between two strings instead of finding exact matches. Our systematic technique could be easily adapted for this situation by properly modifying (3). However, the design proposed in [29] was obtained using dynamic programming approach and has time complexity of $\mathcal{O}(n+m)$. Analytical as well as numerical simulations of our designs show an average time complexity of $\mathcal{O}(n-m)$ (Sections 7 and 8). Our design approach could also be adapted to implement the approximate text searching considered by Michailidis and Margaritis [31], [32].

To summarize, the systematic technique we used to explore possible processor array structures for the string search problem produced novel and efficient designs in addition to all the designs previously proposed in the literature.

10 Conclusion

This paper presented a systematic technique for expressing the string search algorithm as a regular iterative expression to explore all possible processor arrays for the string search algorithms as used in deep packet classification. The computation domain of the algorithm was obtained and three affine scheduling functions were presented. The technique allowed some of the algorithm variables to be pipelined while others are broadcast over system-wide buses. Nine possible processor array structures were obtained and analyzed in terms of speed, area, power, and I/O timing requirements. Time complexities were derived analytically and through extensive numerical simulations. The proposed designs exhibit optimum speed, area, and power. The processor arrays were compared with previously derived processor arrays for the string matching problem. In all designs, we showed that the resulting processor arrays have m processors and their average time to produce a result is $n - m$ (58).

APPENDIX

TIME COMPLEXITY CALCULATION FOR THE AVERAGE CASE

Using (56) and (57), we have

$$
\begin{aligned}
T_{av} &= a^m \sum_{i=0}^{n-m} (m+i)(1-a^m)^{i-1} \\
&= \frac{a^m}{1-a^m} \sum_{i=0}^{n-m} (m+i)(1-a^m)^{i} \\
&= a^m(1+a^m) \sum_{i=0}^{n-m} (m+i)(1-a^m)^{i} \\
&\qquad \left[(1-a^m)^{-1} \approx 1+a^m, \right. \\
&\qquad \left. \text{neglecting higher terms of } a^m\right] \\
&= a^m \sum_{i=0}^{n-m} (m+i)(1-a^m)^{i} \\
&\qquad \left[\text{neglecting higher terms of } a^m\right] \\
&= ma^m \sum_{i=0}^{n-m} (1-a^m)^{i} + a^m \sum_{i=0}^{n-m} i(1-a^m)^{i} \\
&= ma^m \times \frac{1-(1-a^m)^{n-m+1}}{1-(1-a^m)} + \\
&\quad a^m \times \frac{(1-a^m)\left(1-(n-m+1)(1-a^m)^{n-m}+(n-m)(1-a^m)^{n-m+1}\right)}{(1-(1-a^m))^2} \\
&= m - m(1-a^m)^{n-m+1} + \\
&\quad \frac{(1-a^m)-(n-m+1)(1-a^m)^{n-m+1}+(n-m)(1-a^m)^{n-m+2}}{a^m} \\
&= m - m(1-a^m)^{n-m+1} + \\
&\quad \frac{(1-a^m)-(n-m+1)(1-a^m)^{n-m+1}+\frac{(n-m)(1-a^m)^{n-m+3}}{1-a^m}}{a^m} \\
&= m - m(1-a^m)^{n-m+1} + \\
&\quad \frac{(1-a^m)-(n-m+1)(1-a^m)^{n-m+1}+(n-m)(1-a^m)^{n-m+3}(1+a^m)}{a^m} \\
&\qquad \left[(1-a^m)^{-1} \approx 1+a^m, \right. \\
&\qquad \left. \text{neglecting higher terms of } a^m\right] \\
&= m - m(1-a^m)^{n-m+1} + \\
&\quad \frac{(1-a^m)-(n-m+1)(1-a^m)^{n-m+1}+(n-m+na^m-ma^m)(1-a^m)^{n-m+3}}{a^m} \\
&= (m-m) + \frac{1-(n-m+1)+(n-m+na^m-ma^m)}{a^m} \\
&\quad [1-a^m \approx 1, \text{since } 1 \gg a^m] \\
&= \frac{1-n+m-1+n-m+na^m-ma^m}{a^m} = n-m.
\end{aligned}
$$

REFERENCES

[1] A.N.M.E. Rafiq, M.W. El-Kharashi, and F. Gebali, "A Fast String Search Algorithm for Deep Packet Classification," *Computer Comm.*, vol. 27, no. 15, pp. 1524-1538, Sept. 2004.

[2] H.T. Kung and C.E. Leiserson, "Systolic Arrays for VLSI," *Proc. Sparse Matrix Symp.*, pp. 256-282, 1978.

[3] S.K. Rao and T. Kailath, "Regular Iterative Algorithms and Their Implementation on Processor Arrays," *Proc. IEEE*, vol. 76, no. 3, pp. 259-269, Mar. 1988.

[4] S.Y. Kung, *VLSI Array Processors*. Englewood Cliffs, N.J.: Prentice-Hall, 1988.

[5] E.M. M. Abdel-Raheem, "Design and VLSI Implementation of Multirate Filter Banks," PhD dissertation, Dept. of Electrical and Computer Eng., Univ. of Victoria, 1995.

[6] E. Abdel-Raheem, F. El-Guibaly, and A. Antoniou, "Systolic Implementation of FIR Decimators and Interpolators," *IEE Proc. Circuits Device System*, vol. 141, pp. 489-492, Dec. 1994.

[7] M.O. Esonu, A.J. Alkhalili, S. Hariri, and D. Al-Khalili, "Systolic Arrays—How to Choose Them," *IEE Proc.-E Computers and Digital Techniques*, vol. 139, no. 3, pp. 179-188, May 1992.

[8] J.M. D. Y. Wong, "Optimization of Computation Time for Systolic Arrays," *IEEE Trans. Computers*, vol. 41, no. 2, pp. 159-177, Feb. 1992.

[9] F. El-Guibaly and A. Tawfik, "Mapping 3D IIR Digital Filter onto Systolic Arrays," *Multidimensional Systems and Signal Processing*, vol. 7, no. 1, pp. 7-26, Jan. 1996.

[10] M. Nossik, "Optimizing Network Processing with Deep Packet Classification," OPTIMIZING_WP.pdf, http://www.idt.com/docs/, 2002.

[11] S. Iyer, R.R. Kompella, and A. Shelat, "ClassiPI: An Architecture for Fast and Flexible Packet Classification," *IEEE Network*, vol. 15, no. 2, pp. 33-41, Mar./Apr. 2001.

[12] D. Bursky, "Search Engines Take on Larger Forwarding Tables," *Electronic Design*, vol. 51, no. 27, p. 48, 2003.

[13] M. Peyravian, G. Davis, and J. Calvignac, "Search Engine Implications for Network Processor," *IEEE Network*, vol. 17, no. 4, pp. 12-14, July/Aug. 2003.

[14] G.A. Stephen, *String Searching Algorithms*, Lecture Notes Series on Computing, D.T. Lee, ed., Bangor, Gwynedd, UK: World Scientific, vol. 3, 1994.

[15] T. Lecroq, "Experiments on String Matching in Memory Structures," *Software: Practice and Experience*, vol. 28, no. 5, pp. 561-568, Apr. 1998.

[16] J. Jájá, *An Introduction to Parallel Algorithms*, Reading, Mass.: Addison-Wesley, ch. 7, pp. 311-365, 1992.

[17] M. Crochemore, Z. Galil, L. Gasieniec, K. Park, and W. Rytter, "Constant-Time Randomized Parallel String Matching," *SIAM J. Computing*, vol. 26, no. 4, pp. 950-960, Aug. 1997.

[18] U.Z. T. Goldberg, "Faster Parallel String-Matching via Larger Deterministic Samples," *J. Algorithms*, vol. 16, no. 2, pp. 295-308, Mar. 1994.

[19] Z. Galil, "A Constant-Time Optimal Parallel String-Matching Algorithm," *J. Assoc. for Computing Machinery*, vol. 42, no. 4, pp. 908-918, July 1995.

[20] J. Misra, "Derivation of a Parallel String Matching Algorithm," *Information Processing Letters*, vol. 85, no. 5, pp. 255-260, Mar. 2003.

[21] J. Misra, "Powerlist: A Structure for Parallel Recursion," *ACM Trans. Programming Languages and Systems*, vol. 16, no. 6, pp. 1737-1767, Nov. 1994.

[22] K.L. Chung, "$\mathcal{O}(1)$-Time Parallel String-Matching Algorithm with VLDCs," *Pattern Recognition Letters*, vol. 17, no. 5, pp. 475-479, May 1996.

[23] A.A. Bertossi and F. Logi, "Parallel String-Matching with Variable-Length Don't Cares," *J. Parallel and Distributed Computing*, vol. 22, no. 2, pp. 229-234, Aug. 1994.

[24] Y. Takefuji, T. Tanaka, and K.C. Lee, "A Parallel String Search Algorithm," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 22, no. 2, pp. 332-336, Mar./Apr. 1992.

[25] H.D. Cheng and K.S. Fu, "VLSI Architectures for String Matching and Pattern Matching," *Pattern Recognition*, vol. 20, no. 1, pp. 125-141, 1987.

[26] M.E. Isenman and D.E. Shasha, "Performance and Architectural Issues for String Matching," *IEEE Trans. Computers*, vol. 39, no. 2, pp. 238-250, Feb. 1990.

[27] A.V. Aho and J.D. Ulman, *Principles of Compiler Design*, Reading, Mass.: Addison-Wesley, pp. 91-94, 1977.

[28] M.J. Foster and H.T. Kung, "The Design of Special-Purpose VLSI Chips: Example and Opinions," *Proc. Seventh Ann. Symp. Computer Architecture, Int'l Conf. Computer Architecture*, pp. 300-307, May 1980.

[29] A. Mukherjee, "Hardware Algorithms for Determining Similarity between Two Strings," *IEEE Trans. Computers*, vol. 38, no. 4, pp. 600-603, Apr. 1989.

[30] J.H. Park and K.M. George, "Efficient Parallel Hardware Algorithms for String Matching," *Microprocessors and Microsystems*, vol. 23, no. 3, pp. 155-168, Oct. 1999.

[31] P.D. Michailidis and K.G. Margaritis, "Parallel Architecture for Flexible Approximate Text Searching," *CD-ROM Proc. Seventh WSEAS Int'l Multiconf. Circuits, Systems, Comm. and Computers (WSEAS-CSCC 2003)*, July 2003.

[32] P.D. Michailidis and K.G. Margaritis, "Bit-Level Processor Array Architecture for Flexible String Matching," *Proc. First Balkan Conf. Informatics (BCI 2003)*, pp. 517-526, Nov. 2003.

[33] K.R.R. Sastry and N. Ranganathan, "CASM—A VLSI Chip for Approximate String-Matching," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 17, no. 8, pp. 824-830, Aug. 1995.
[34] P.R. Panda, N.D. Dutt, and A. Nicolau, "On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems," *ACM Trans. Design Automation of Electronic Systems,* vol. 5, no. 3, pp. 682-704, July 2000.
[35] F. Elguibaly, "A Fast Parallel Multiplier-Accumulator Using the Modified Booth Algorithm," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing,* vol. 47, no. 9, pp. 902-908, 2000.
[36] F. Elguibaly, "Merged Inner-Prodcut Processor Using the Modified Booth Algorithm," *Canadian J. Electrical and Computer Eng.,* vol. 25, no. 4, pp. 133-139, 2000.
[37] N.H.E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design.* Addison-Wesley, 1992.

Fayez Gebali received the BSc degree in electrical engineering (first class honors) from Cairo University, the BSc degree in mathematics (first class honors) from Ain Shams University, and the PhD degree in electrical engineering from the University of British Columbia where he was a holder of the NSERC postgraduate scholarship. Dr. Gebali is a professor of computer engineering and associate dean of engineering at University of Victoria. His research interests include processor array design for DSP, computer communications, computer arithmetic, and network processors design. He is a senior member of the IEEE Computer Society.

A.N.M. Ehtesham Rafiq received the BSc and MSc degrees in computer science and engineering from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh in 1997 and 2000, respectively. He is currently a PhD candidate in the Electrical and Computer Engineering Department of University of Victoria, Canada. His research interests include computer communications, computer architecture, and VLSI design.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.