Sequential and Parallel Algorithms for Network Packet Classification

by

A. N. M. Ehtesham Rafiq

B.Sc., Bangladesh University of Engineering and Technology, 1997
M.Sc., Bangladesh University of Engineering and Technology, 2000

A Thesis Submitted in Partial Fullfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

©A. N. M. Ehtesham Rafiq, 2006
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Sequential and Parallel Algorithms for Network Packet Classification

by

A. N. M. Ehtesham Rafiq

B.Sc., Bangladesh University of Engineering and Technology, 1997
M.Sc., Bangladesh University of Engineering and Technology, 2000

Supervisory Committee

Dr. Fayez Gebali (Dept. of Electrical and Computer Engineering)
Supervisor

Dr. Amirali Baniasadi (Dept. of Electrical and Computer Engineering)
Department Member

Dr. Issa Traoré (Dept. of Electrical and Computer Engineering)
Department Member

Dr. Ahmed Sourour (Dept. of Mathematics and Statistics)
Outside Member

Supervisory Committee

Dr. Fayez Gebali (Dept. of Electrical and Computer Engineering)
Supervisor

Dr. Amirali Baniasadi (Dept. of Electrical and Computer Engineering)
Department Member

Dr. Issa Traoré (Dept. of Electrical and Computer Engineering)
Department Member

Dr. Ahmed Sourour (Dept. of Mathematics and Statistics)
Outside Member

Abstract

A network processor unit (NPU) is a programmable device that consists of several hardware accelerators for wire-speed networking operations. One of the most important functional units in an NPU is packet classification unit (PCU) that classifies data packets based on single or multiple fields of packet header or contents in payload data. Large number of tasks in computer communication require packet classification. Network packet classification requires two types of matching techniques: (i) exact and (ii) inexact match. There are two solutions for exact match: (i) sequential and (ii) parallel solutions. Inexact match can be of two types: (i) Longest prefix match and (ii) Best match. This dissertation talks about these four techniques required for the PCU.

For the sequential solution, we propose a string search algorithm that requires reduced time complexity. It also requires a small amount of memory, and shows better performance than any other related algorithms as proved by numerical analysis and extensive computer simulations.

For parallel solution, we present a systematic technique for expressing the string search algorithm as a regular iterative expression to explore all possible processor

arrays. 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. The proposed designs exhibit optimum speed and area complexities.

The parallel solution requires an embedding technique that embeds a source processor array onto a target processor array having smaller number of processing elements (PE) to meet the hardware resource constraint. We propose a novel embedding technique. Through numerical analysis and extensive computer simulation, it is proved that the performance of the target array shows the same performance as the source array.

For Longest prefix match (LPM), we propose a novel variable-stride multi-bit trie data structure for IP-lookup table to assist fast IP-lookup and fast lookup table update. In this dissertation, we first explicitly elaborate the solution of a problem in expanding IP (internet protocol) addresses. Through extensive computer simulation on several routing tables, it is proved that our proposed algorithm shows better performance (lookup and update time) than existing algorithms. However, our proposed technique requires larger memory than others. But the memory requirement is quite acceptable considering the current memory availability and price.

We propose a novel Best Match technique required to detect best-matched English words of obfuscated spam words. We have used a non-deterministic finite automaton (NFA) to build the English dictionary. We have used dynamic programming with state pruning to detect the best-matched word of an obfuscated spam word in the NFA. We have done extensive numerical simulations to prove the accuracy of our proposed system. Our system can detect best-matched words of the words obfuscated by spammers using five different techniques: insertion, deletion, substitution, transpose, and word boundary. Upto our knowledge, no other system can deal with all these obfuscating techniques so quickly as ours.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

NPU	Network Processor Unit
PCU	Packet Classification Unit
PE	Processing Element
LPM	Longest Prefix Match
IP	Internet Protocol
QoS	Quality of Service
ASIC	Application Specific Integrated Circuit
MAC	Media Access Control
CAM	Content Access Memory
PCAM	Pair-bit Content Access Memory
RAM	Random Access Memory
SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory
I/O	Input-Output
PE	Processing Element
FPGA	Field Programmable Gate Array
NFA	Non-deterministic Finite Automata
FSA	Finite State Automata
PHY	PHYsical interface
OC-X	Optical Carrier (Max. speed = 51.84X Mbit/s)
SFC	Single-Field Classification
MFC	Multi-Field Classification
DPC	Deep Packet Classification
URL	Uniform Resource Locator
TCP	Transmission Control Protocol

CPU	Central Processing Unit
ACL	Access Control List
ASCII	American Standard Code for Information Interchange
VLSI	Very Large Scale Integration
CRCW	Concurrent Read Concurrent Write
PRAM	Parallel Random Access Machine
EREW	Exclusive Read Exclusive Write
RIA	Regular Iterative Algorithm
DG	Dependence Graph
ALU	Arithmetic and Logic Unit
CMOS	Complementary Metal Oxide Semiconductor
LSGP	Locally Serial Globally Parallel
LPGS	Locally Parallel Globally Serial
GF	Galois Field
LC	Logic Cell
LC-trie	Level Compressed trie
CIDR	Classless Inter-Domain Routing
PL	Prefix Length
HMM	Hidden Markov Model

List of Symbols

T	*Text*
P	*Pattern*
n	Length of T
n_i	i-th node of a trie
m	Length of P
w	Size of data-bus
Σ	Alphabet set
d_1	Distance between two characters
d_2	Distance between two strings
C_1	Number of iterations
C_2	Number of characters
T_w	Worst case time complexity
T_b	Best case time complexity
T_a	Average case time complexity
$C_{i,j}$	TRUE if $t_i = p_j$; FALSE if $t_i \neq p_j$
p_m	Match probability between two characters
p_{mm}	Mis-match probability between two characters
Δ	Difference between two execution times
a, b, c	Number of iterations
d	number of input streams
Y	Output variable
y_i	Output variable at instance i
$t(p)$	Scheduling function–associates time t to a point $\mathbf{p}$

$\mathbf{s}$	Scheduling vector
s	Constant
$\mathbf{e}$	Null vector
$\mathbf{P}$	Projection matrix
$\mathbf{d}$	Projection direction
$\mathbf{p}$, $\mathbf{q}$, $\mathbf{r}$	Point vector
$\mathbf{a}$	Vector of available PEs
a	Available PE
τ_{clk}	Clock period
τ_p	Processing delay
τ_d	Output driver delay
τ_b	Bus delay
τ_m	Memory access delay
C_l	Load capacitance
C_g	Gate capacitance
$\varrho(x)$	Power consumption for design x
ϱ_{PE}	Power consumption by PE
ϱ_b	Power consumption of driving the bus
Q	Queue
A_{ij}	Result produced by PE_i and propagated to PE_j
$p_{i,j}$	Transition probability from state i to state j
$\mathbf{A}$	Transition matrix
$\mathbf{x}(\eta)_z$	Probability of being in state z at time step η
C_t	Number of time steps

e	Routing entry
P_i	multiple bits of i-th stride
p_{16}	Probability of being in $Level16$
p_{CQ16}	Probability of being in $LevelCQ16$
p_{24}	Probability of being in $Level24$
p_{CQ24}	Probability of being in $LevelCQ24$
p_{32}	Probability of being in $Level32$
p_{CQ32}	Probability of being in $LevelCQ32$
T_{lookup}	Time complexity for table lookup
T_{remove}	Time complexity to add an entry to the lookup table
T_{add}	Time complexity to remove an entry from the lookup table
Γ	Alphabetic characters
Ψ	Characters look-alike alphabetic characters
Φ	Non-alphabetic characters look-alike alphabetic characters
Λ	All characters
F	Flag
S, S_1, and S_2	String
$\mathcal{P}$	Array of pointers
T_1	Hash table
T_2	Table for preprocessing for Best Match
A	Address
Θ	Threshold

s_i	i-th state of NFA
$\mathcal{W}$	Best-matched word
$\mathcal{C}_i$	Cost to find W_i
DP	Dynamic Programming
$\mathcal{S}$	List of next states for a particular input in the NFA
SL	List of all next states in the NFA
L	Search for longer match
L'	Search for low cost
W	Check word boundary
W'	Don't check word boundary
R	Use rule set
R'	Don't use rule set
V_a	Average number of visited states
L	Number of visited states

Acknowledgments

All praise be to the Almighty God who has given me knowledge, skill, patient, and perseverance to finish my works for the Ph.D. dissertation.

It is a great opportunity to remember my parents for their love and mental supports all through my Ph.D. works. This acknowledgment is nothing compared to what they did for me to grow up and acquire my background knowledge to accomplish the works for the Ph.D. dissertation.

There is no word to praise my supervisor, Dr. Fayez Gebali. His scholarly advices and fatherly guidance helped me to finish my Ph.D. dissertation works. His continuous inspiration and financial support helped me to concentrate on the research work. My stay under his supervision also helped me to increase not only my research ability, but also my teaching and managerial abilities for big events.

I have got love, scholarly discussion, and tireless inspiration from my wife throughout the tenure of my Ph.D. studies. As a gift of my hard-earned achievement, she is going to give birth our first baby, Nabhan by the grace of Almighty. Here I also recognize the help, support, and love during my final days of my Ph.D. dissertation from other family members: Salam Azad, Rokhsana, Sabik, and Radia.

I like to acknowledge the advice I received from my supervisory committee members: Dr. Issa Traoré, Dr. Amirali Baniasadi, and Dr. Ahmed Sourour to make my dissertation complete and resourceful.

Its my pleasure to give special thanks to Watheq El-Khrarashi and Muhammad Marsono for their invaluable support and inspiration in my research works. However, it would be unfair, if I do not mention the names of my other colleagues: Khalid Khayyat, Mohamed Fayed, Abdelsalam Amer, Haytham El-Miligi, Ahmad Abdullah, Esam Khan, and Tarek Nasser.

Finally, I remember departmental staffs: Moneca Bracken, Vicky Smith, Sarah Jenkinson, Lynne Barrett, Mary-Anne Teo, Erik Laxdal, and Steve Campbell for their supports during my stay in the university.

Dedication

To my parents.

Chapter 1

Introduction

Today's communication networks need huge amount of data processing at very high speed. Networking devices have also to be adaptable to ever-changing protocols with QoS management capabilities. Different techniques are being tried to address these issues. To support changing protocols, a flexible device is required. For fast data processing, a device with the fast processing capability is required. Figure 1.1 shows different techniques used in the communication network. Standard old-fashioned switch-fabric technique is used for the data forwarding only. It cannot be programmed to dynamically adapt different protocols. ASIC (Application specific integrated circuit) is built for specific purposes and so it shows better performance. But ASIC is not flexible enough to be used for other purposes. Some people like general purpose processors as they are flexible enough to be used for any purposes. But they have no special hardware to speed up the performance. NPU (Network Processor Unit) [1–25] has both ASICs and general purpose processors in it to achieve ultimate performance and flexibility that is required for today's data transmission. Thus NPUs are the right choice for huge data processing at wire-speed.

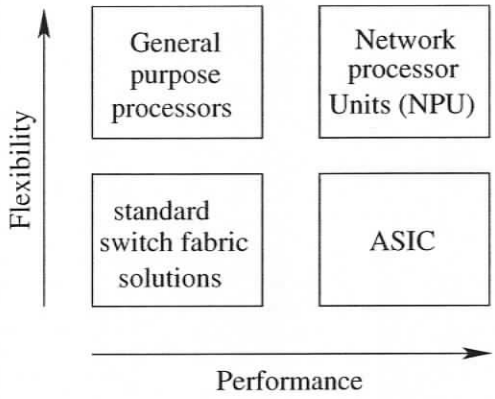


Figure 1.1: Communication design matrix.

1.1 A Generic NPU

Although the structure of an NPU varies by its intended uses, a generic structure of an NPU can be drawn as shown in Figure 1.2. A generic NPU should have at least following components:

Memories are for data storage, queue, and look-up table entries. Research is required to determine the types of memories to be used for those purposes and to determine whether the memories should be on-chip or off-chip. Especially the look-up table needs intelligent memory organization to take advantage of the look-up table search algorithm to speed up the search operation for next hop's addresses [26].

PCU classifies packets based on packet header and/or packet data. It incorporates (i) a parser to identify packet format, (ii) a string search machine for content inspection, and (iii) a look-up table for IP routing. Lots of functions of an NPU depends on classification results of the PCU. So, fast and flexible classification unit is essential for an NPU. Research is required for (i) a fast and efficient string search algorithm for content inspection and (ii) a look-up table based

search algorithm for next-hop-address searching.

Queue management unit controls packet flow and manages the packet buffer. Research is required to devise an efficient flow control algorithm.

Control unit, a programmable device, controls all the components of the NPU. Research is required to find the instruction set and instruction format for this control unit. It is also to be decided whether the control unit controls other component through point-to-point connection or through system bus.

Bus carries data and/or control signals. A fast and efficient system bus is very important for wire-speed data processing. Research is required to determine the organization and characteristics of the bus.

The generic NPU in Figure 1.2 works in the following way. Packet (up to layer 2) arrives through external PHY (PHYsical interface) and framer unit. Queue management unit puts the incoming packet on the packet buffer. Parse unit of PCU recognizes the packet header and forwards the required information to Look-up table. Look-up unit of the PCU searches next hop addresses of the packets with the help of the look-up table [26]. While table look-up unit works, the string search engine of the PCU may search some strings in the packet's payload data for security or other reasons to speed up the overall data processing. Classified packet is then treated by the control unit and/or other optional and additional units based on policies defined by users. Finally the packet is forwarded to the next hop by the Queue management unit through the PHY and framer unit.

Thus one of the most important units of an NPU to be explored is packet classification unit (PCU) [13, 27–31]. A PCU classifies a packet based on its one or more header fields or specific contents in payload data. Packet classification is essential for routing, security, differentiated service, QoS management, etc. Thus an efficient

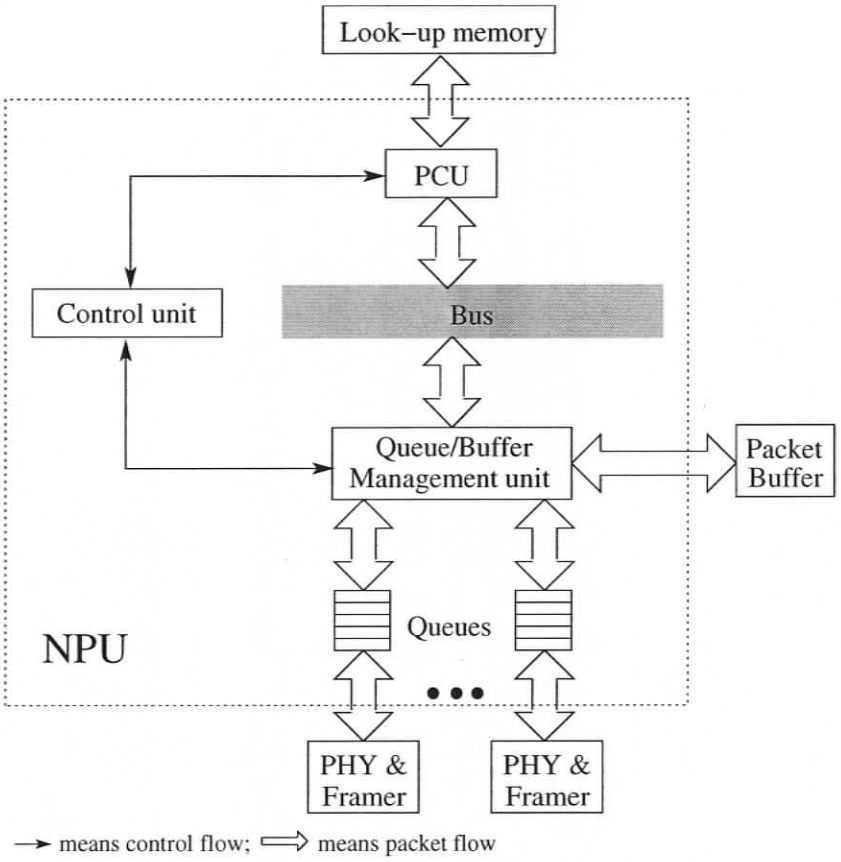


Figure 1.2: Generic architecture of a NPU.

and fast classification unit is required for an NPU to offload the time-consuming classification task and thereby cope with the increasing complexities of the computer network.

1.2 Packet Classification Unit

Internet is being extensively used now-a-days by people of different tastes and requirements. Some people use it for entertainment, some use it for secured data, whereas some use it for learning. These different uses require different network resources

and rules for efficient transmission. So, we have to classify network data packets. Previously packet forwarding decision was made on only level two (MAC) and level three (IP) header; whereas today we have to even check layer seven (pay load) data for content inspection [32]. Moreover, internet speed of OC-192 (10 Gbps) is already achieved and OC-768 (40 Gbps) speed is within sight. The necessity of this complex and fast packet classification requires a dedicated PCU in every NPU.

1.2.1 Types of Classification

The packet classification unit does three types of classification on networking data [27, 33].

1. *Single-field (SF)* classifier looks at a single field in the packet header for classification. The mostly-used layer 2 (MAC) and layer 3 (IP) routing fall in this classification. Usually look-up table based classification is sufficient for these operations. In this classification, one fixed field is extracted from the packet header and its value is usually searched in a look-up table. The packet is classified according to this result.

2. *Multi-field (MF)* classifier looks multiple fields at non-contiguous locations to classify packets. This process makes a complicated search key for the look-up table. Network switches with features such as DiffServ require this type of classification [RFC 2475]. Only look-up table based classification is not sufficient in this case, a dedicated classification unit is also required.

3. *Content inspecting* classifier examines packet content (payload data) in order to classify packets [31, 34–38]. The search key is irregular and may consist of hundreds of characters. Look-up concept is not suitable here, instead a string search based classification unit is required. One of the many important

applications of this classification is security.

1.2.2 Classifier Applications

There are many uses of packet classification unit in data networking. Several important examples are listed below.

Routing: Layer 2 (MAC) and layer 3 (IP) routings are used in almost every networking device. Previously, these are only single-field classification applications, but with the added complexity of networking protocols, now complex multi-field routing is required. A dedicated device will make this routing faster [39].

Increased Server Efficiency: Retrieving information from local memory is many times faster than retrieving it from remote server. Classifier can help to store frequently visited URLs in the server cache and thus improves server efficiency and performance [34].

Differentiated Service: Different users require different services, different QoS, etc. Classifiers can help to interpret the visitor and information content and give value-added, customized, differentiated QoS services to different customers [34].

Persistence: In many cases, clients must retrieve multiple web objects from the same server. For example, in on-line shopping, a client must constantly talk to the same real server for the duration of the transaction, which typically spans multiple TCP connections. In some cases, several servers may be required to fulfill a particular request. In both cases, The reliable way to identify the user through cookies. Cookies are embedded inside the payload data. The string search unit of the PCU help to check cookies at a very fast rate to be persistent in both cases [34].

Server Load Balancing: This improves network performance and resource utilization significantly by distributing traffic efficiently so that individual server is not overwhelmed by sudden fluctuations in activity. It examines all user traffic and, when appropriate, redirects traffic to the appropriate server according to centrally defined policies. Load balancer can also works as a web switch [34,39,40].

Security: Classification unit can provide security in data communication in various ways. It can detect and stop intruders. It can detect viruses communicating in the networking path. It can be used to accept or deny packets according to the given policy. The classifier checks the packet header and payload data to provide these security issues [39,40].

1.3 Motivation

In this section, we talk about the justification of my research work plan. We divide this section into two subsections: (i) the importance of NPU is described in Section 1.3.1 and (ii) the importance of PCU (a part of NPU)is given in Section 1.3.2.

1.3.1 Necessity of NPU

For wire speed data transmission, the conducting processor has to process large amount of data in real time. General purpose processors do not have this capability. we list the reasons of implementing NPUs in the following.

Parallel Processing: NPUs employ multiple functional units in parallel to process huge amount of data at wire-speed. we would like to give a simple calculation in this regard in Example (1).

Example 1: Following assumptions are taken for the calculation:

- One IPv6 packet [41] is received at a time (i.e. single I/O).
- Single CPU is employed.
- The CPU has only one functional unit (i.e. not superscalar).
- A typical packet processing, by the assumed CPU, consists of 6 ($= N$) tasks: (i) assembling, (ii) classification, (iii) header modification, (iv) segmentation, (v) scheduling, and (vi) forwarding. Each task gets equal time sharing from the CPU.
- line rate, $L = 10$ Gbps (OC-192).
- CPU clock speed, $r = 2$ GHz.
- CPU word size, $w = 128$ bit (data path size of Intel Pentium 4 [42]) .
- Packet size, $p = 64$ byte.

Maximum time allowed for packet transmission,

$$T = \frac{p}{L} = \frac{64 \times 8}{10 \times 10^9} = 51.2 \text{ ns}$$

Maximum cycles allowed for packet processing by the assumed CPU,

$$P_c = T \times r = 51.2 \times 10^{-9} \times 2 \times 10^9 = 102.40 \approx 103 \text{ cycles.}$$

Each task gets maximum

$$ET_c = \frac{P_c}{N} = \frac{103}{6} \approx 17 \text{ cycles.}$$

To process a packet, each task has to be done,

$$PET = \frac{p}{w} = \frac{64 \times 8}{128} = 4 \text{ times.}$$

Each task has to be completed within

$$TC_c = \frac{ET_c}{PET} = \frac{17}{4} \approx 4 \text{ cycles.}$$

Thus a 2 GHz CPU has to complete each of six processing on a 64-byte sized packet at OC-192 line rate within 4 cycles. It is impossible for a single CPU to complete all the processing including memory operations within the time limit. This scenario is for single I/O. Multiple I/Os would make the condition worse. So, we need a special unit, NPU, having multiple functional units which can work parallel for wire-speed data transmission. ■

Multiple I/Os: For large data transmission, several high-bandwidth input, output and memory access events must be supported by the processor concurrently. Since single general-purpose CPU cannot quickly multiplex between various concurrent tasks, a general-purpose CPU might not achieve line-rate performance beyond OC-12 [43]. For this reason, we need NPU with fast multiplexing ability between multiple inputs, outputs and memories, instead of a general-purpose CPU.

Fast I/Os: Communication processors must have fast I/O system. Conventional DMA and interrupt mechanisms cannot support line rate more than OC-48 [43]. Also, there must be some mechanisms so that input packet can be started processing without any delay and after processing outgoing packets can be sent to appropriate output in zero time.

Instruction Cache: Generally networking packets are random in nature. This would make data cache inefficient. But, as same instructions have to be executed on different data, instruction cache might be helpful. Thus traditional cache mechanism would not be effective in NPU.

Special Instruction Set: NPUs have spacial instruction set for data communication. This helps in two ways: (i) instruction set becomes smaller that makes

memory requirement lower and (ii) programmability becomes easier with application specific instructions.

Special Memory Units: NPUs have specially organized memory units like look-up table, queue, etc. These memory units help fast data processing operations in NPUs.

From above discussions, it is clear that we need a special device, NPU, for wire-speed data transmission.

1.3.2 Necessity of Dedicated Packet Classifier in NPU

The first network processing solutions emerged as highly integrated functional ICs called Network Processors, which were designed to manage all network processing functions, including classification. But as the data rates and data volume in the network continue to grow, surpassing the scaling capabilities of semiconductor devices, it becomes obvious that a dedicated classification unit is required. The classification unit off-loads the classification works from NPU in order to cope with the growing demand for speed. We list below the reasons of having a packet classification unit in NPU [31].

Task Dependency: Since basic tasks of today's NPU (e.g., routing, QoS maintenance, scheduling, security, and so on) depend on the classification of packets, a special hardware is required to classify packets quickly so that other data processing tasks can get classified packets in no time. Thus the PCU can help NPUs to achieve wire-speed performance.

Task Off-loading: Classifier Off-loads classification job from NPU. NPU is benefited by this off-load in two ways: (i) increased speed and (ii) minimized code space.

We give examples of increased speed in Example (2) and of minimized code space in Example (3).

Example 2: (Increased speed) We calculate the search time of the IP address of the packet header in look-up table. We assume the following values and notations for the calculation:

- Intel IXP2800 NPU works on IPv6 packets [6, 41].
- IP address size (Bits of interest), $n = 128$ bit.
- Packet header size, $k = 40$ byte.
- Packet size, $p = 64$ byte.
- Number of processor engines (PEs) in NPU, $E = 16$.
- PE's word size, $w = 32$ bit.
- PE's clock rate, $r = 1.4$ GHz.
- Line rate, $L = 10$ Gbps (OC-192).
- RAM speed, $s = 133$ Mhz (PC-133). It is assumed that RAM has one port.
- Memory operation takes only transfer time. Access time, latency time, preparation time, etc. are assumed zero.
- Hashing is used for look-up table. Collision is not considered for simplicity.
- Hash function requires $h = 64$ bit to calculate the key.
- Hash function calculation requires $h_c = 12$ cycles.
- Comparison between h bit data requires, $c_c = 1$ cycle.
- Modification time on packet header, $m_c = 2$ cycles.

According to the assumptions, one operation on RAM takes,

$$r_c = \frac{r}{s} = \frac{1.4 \times 10^9}{133 \times 10^6} \approx 11 \text{ cycles.}$$

The typical IP search time,

$$\begin{aligned} t &= \text{reading and modification time of packet header from memory} + \text{hash} \\ &\quad \text{calculation time of the IP address} + \text{comparison time of hash result} \\ &\quad \text{with look-up entries} \\ &= t_1 + t_2 + t_3 \end{aligned}$$

Processing starts with reading a packet's protocol headers for parsing the key. The time to read the packet header from memory,

$$\begin{aligned} t_1 &= (\text{packet header/PE's word size}) \text{ reads} \times \text{cycles for} (\text{One application read} + \\ &\quad \text{one packet read} + \text{application execution on packet header}) / \\ &\quad \text{PE's clock rate} \\ &= \frac{k}{w} \times \frac{r_c + r_c + m_c}{r} \\ &= \frac{40 \times 8}{32} \times \frac{11 + 11 + 2}{1.4 \times 10^9} \\ &= 171.43 \text{ ns} \end{aligned}$$

The time to calculate the hash value, ·

$$\begin{aligned} t_2 &= \text{number of hash operations} \times \text{cycles for hash operation/PE's clock rate} \\ &= \frac{n}{h} \times \frac{h_c}{r} \\ &= \frac{128}{64} \times \frac{12}{1.4 \times 10^9} \\ &= 17.14 \text{ ns} \end{aligned}$$

After the calculation of the hash function, this value is compared with the hash entry. The time requirement,

$$\begin{aligned} t_3 &= \text{number of hash operations} \times \text{cycles for (one application read + comparison time of } h \text{ bit) /PE's clock rate} \\ &= \frac{n}{h} \times \frac{r_c + c_c}{r} \\ &= \frac{128}{64} \times \frac{11+1}{1.4 \times 10^9} \\ &= 17.14 \text{ ns} \end{aligned}$$

Therefore, total time for classification,

$$t = t_1 + t_2 + t_3 = 205.71 \text{ ns.}$$

But total time available to process a packet is

$$\begin{aligned} T &= \text{packet size} \times \text{number of PEs/line rate} \\ &= \frac{64 \times 8 \times 16}{10 \times 10^9} = 819.20 \text{ ns.} \end{aligned}$$

Thus classification takes

$$\frac{t}{T} \times 100\% = \frac{205.71 \times 100}{819.20} \approx 25\%$$

of total packet processing operations. In this way, classifier offloads around 25% works of NPU. ■

Example 3: (Minimized code space) We assume the following for the calculation:

- Total number of instructions in the NPU, $I = 3000$.
- Number of instructions for classification, $i_c = 400$.

- A dedicated classifier produces classification results in the form of a tag. The NPU requires $i_t = 30$ instructions to read and process this tag.

Thus a dedicated classifier can save

$$\frac{(i_c - i_t)}{I} \times 100\% = \frac{(400 - 30) \times 100}{3000} \approx 12\%$$

code space. ■

1.4 Problem Formulation

Packet classification requires two types of matching techniques: (i) exact and (ii) inexact matching as drawn in Figure 1.3. There are two solutions for exact matching: (i) sequential solution and (ii) parallel solution. Inexact matching can also be of two types: (i) longest prefix match and (ii) best match. In the following five sec-

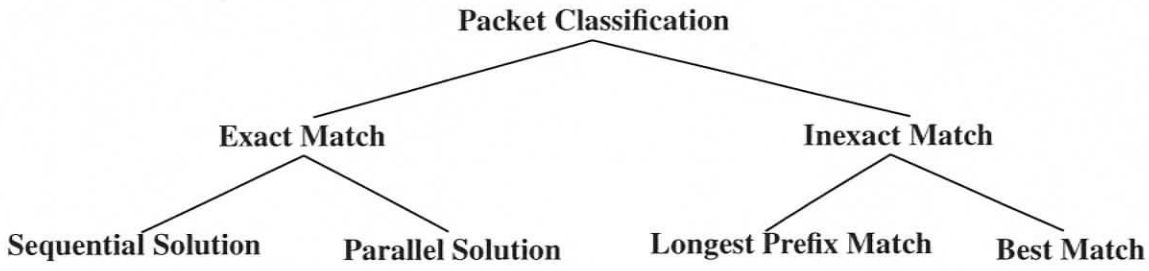


Figure 1.3: Taxonomy of packet classification.

tions, we formulate five problems to be solved in this dissertation for four matching techniques required for PCU.

1.4.1 Exact Match: Sequential Solution

Exact match operates on a given alphabet set Σ of size $|\Sigma|$, a pattern $P = p_0 p_1 \cdots p_{m-1}$ of length m, 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.

Exact match is required for differentiated service, server load balancing, security and so on. Among three types of classification (Section 1.2.1), content inspection requires most time-consuming exact matching technique. Classification by content inspection is sometime referred as deep packet classification. The need for deep packet classification is increasing rapidly with the emerging content-aware applications, such as content-switching, load balancing, streaming data, policy-based firewalls, etc. Such applications are characterized by long T and P, and irregular structures of P. For long T and P, the memory usage is higher. So, the exact match algorithm should utilize memory intelligently to reduce memory requirements. Also, to speed up the operation, a large number of bits (w), called an alphabet, are compared in parallel for exact match. Larger number of parallel bits make the alphabet set ($|\Sigma| = 2^w$) larger. Thus we need a string search algorithm that is fast for large n, m, and $|\Sigma|$ and at the same time simple and requires small memory which makes it suitable for hardware implementation.

Traditional look-up table is not suitable for deep packet classification. Look-up tables [26, 38, 44–47] require a pre-specified type of data, the data has to be organized in specific order to search regular P. For these reasons, the string search algorithms are well-suited for the fast deep packet classification [38, 48].

Different network search operations can be done by content-addressable memory (CAM)-based search algorithms [49–51]. These algorithms can search a P in parallel, irrespective of the length of the search table. However, the delay encountered depends on the level of parallelism employed and the hardware details. CAM has some potential problems e.g. (i) CAM is not suitable for the irregular *patterns*, (ii) it is more costly, (iii) it dissipates more heat, (iv) it consumes more power, and (v) it requires more silicon area than other conventional memory [50, 52, 53]. Thus CAM is suitable for the case where search time is very critical, P is regular, and the

memory requirement is very low such as QoS, access control list (ACL), etc. [48]. That means, CAM is not suitable for such networking operations like deep packet parsing that requires large databases [38]. For applications requiring large databases, it is more cost-effective to use an algorithmic search technique.

From above discussions, we conclude that an algorithmic search or string search algorithm can perform the fast irregular P search in the long P. In Chapter 2, we propose a string search algorithm suitable deep packet classification.

1.4.2 Exact Matching: Parallel Solution

Several efficient sequential string search algorithms have been developed [54–56]. The average time complexity for implementing the string search problem on a single processor was proved to be $\mathcal{O}(n)$ (Chapter 2). Also 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 parallel implementation. 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 [57–59]. Several authors developed techniques for mapping regular iterative algorithms onto processor arrays [58–64]. Chapter 3 presents a systematic methodology for obtaining several processor array architectures for deep packet classification based on the techniques developed in [64].

1.4.3 Embedding Technique for Parallel Solution

The literature abounds with design techniques of processor array structures for implementing several recursive algorithms. Some well-known works of the design of

processor arrays are described in [57–59, 64, 65]. But there is no discussion on how to adapt the number of processing elements (PEs) to meet hardware implementation constraints without degrading the system performance.

In [65], we presented a systematic methodology for obtaining several processor array architectures for the string search algorithm. This methodology guarantees that the workload of each PE of the processor array is one per time step. But this methodology, along with other methodologies proposed so far, produces processor arrays in which the number of PEs depends on the algorithm parameters. The main disadvantage of these methodologies is that different-sized processor arrays are produced each time the algorithm parameters change. Also the resulting processor array size might exceed the hardware resource constraints. The feasible solution of these problems is to embed the input-dependent processor arrays (source) into the input-independent (i.e., fixed-sized) processor array (target). The target array may have degraded performance because of:

- workload increase of each PE,
- complex interconnection among PEs,
- increased memory access, and
- multiplexed I/Os.

In [66], we presented four embedding techniques: (i) modulus operation, (ii) dynamic mapping, (iii) probabilistic mapping, and (iv) changing granularity. In Chapter 4, we propose a novel embedding technique based on the probabilistic mapping and apply the technique on the string matching algorithm for deep packet classification (DPC) to prove that the performance of the target array does not degrade compared to the performance of the source array.

1.4.4 Longest Prefix Match

Longest prefix match (LPM) is primarily used to determine the best next_hop route for a packet based on its destination address. A typical router has a lookup table which consists of prefixes (IP addresses) and next_hops as shown in Table 1.1, where the $\star$ is a wild-card character, i.e., the string is followed by any number of 0s and 1s. In a lookup table, one destination address may match multiple table entries. In such cases, the best next_hop would correspond to most specific prefix or longest prefix match. For example, assume a packet's destination address is 101010$\star$ which matches

Table 1.1: A sample lookup table. $\star$ is a wild-card character.

Prefix	Next hop
0010$\star$	1
010$\star$	2
011$\star$	3
10100$\star$	4
10101$\star$	5
101010$\star$	6
11$\star$	7

two entries in Table 1.1. Among these two, the most specific or longest prefix match produces next_hop 6. The LPM is the most time-consuming task of IP-routing. A new IP-addressing scheme, namely Classless inter-domain routing (CIDR), compounds the complexity of the LPM problem [67]. In Chapter 5, we address the LPM problem to make fast IP-routing considering every single possibility in the routing entry.

1.4.5 Best Match

Unsolicited and unwanted e-mails are called spams [68]. Spam is not just an annoyance. Substantial amounts of time, effort, and money are being wasted on spam [69–71]. Optimistic estimates say that on the average, a USA employee gets 15-20 spam emails daily. According to a CNN article, a study at the University of Maryland says the loss of productivity from spam is costing USA companies about $22 billion per year [72]. This loss is due to the time, estimated at 3 minutes per day, spent in classifying emails and trashing unwanted ones. A recent study also gives the similar amount of loss for spams [73].

Content-based spam filtering is one of the well-known techniques [74]. There are two major tasks for the content-based spam filtering: word or token extraction and spam filtering or packet classification. After extracting the most interesting tokens from an e-mail, Bayesian filter [75], genetic algorithms [76], or neural networks [77] uses the tokens to decide the spam probability of the e-mail message. But the problem is that as people are getting and blocking spams, the spammers are becoming cunning day by day [78]. They obfuscate the well-known spam words in different ways to circumvent the spam filtering strategies [79]. If obfuscated words can be replaced by the best-matched English words before filtering, then the filtering techniques would perform better.

Words can be obfuscated in five different ways as given in Table 1.2 [80]. If we look at the Table 1.2 carefully keeping in mind that spammers want to convey their message to a human reader without much confusion, we can conclude that spammers are allured to use insertion, substitution, and word boundary most to obfuscate spam words, because words obfuscated by these three techniques can convey the message without much confusion. On the other hand, spammers generally shy away from transpose because it is hard to recognize the word obfuscated by transpose (Proved

Table 1.2: Different obfuscating techniques.

Obfuscating technique	Original word	Obfuscated word
insertion	offer	offfer
deletion	interest	intrest
transpose	treat	traet
substitution	loan	l0@n
word boundary	price off	Price_Off

in Section 6.7). For this reason, for obfuscated spam words, we assume transpose as two substitutions, i.e., we present here a technique for Best Match that correct four types of errors (insertion, deletion, substitution, and word boundary) instead of five as mentioned in Table 1.2. Chapter 6 describes a novel Best Match technique required to detect best-matched English words of obfuscated spam words.

1.5 Contributions

We have five major contributions as described below:

1.5.1 Exact Match: Sequential Solution

We propose a string search algorithm that requires reduced time complexity. It also requires a small amount of memory, and shows better performance than any other algorithm for deep packet classification based on their payload data. The proposed algorithm is based on Boyer-Moore algorithm but requires a much reduced number of operations. In addition, our algorithm's memory requirement is lower than Boyer-Moore algorithm without sacrificing its speed. We have done time complexity analysis and verified its time complexities through extensive numerical simulations.

These simulations show that our algorithm's performance is better for long text, long pattern, and large alphabet set and even its worst case time complexity linearly depends on the length of the text. These works are published in [55, 81].

1.5.2 Exact Match: Parallel Solution

We propose a systematic technique for expressing the 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. These works are published in [65, 82, 83].

1.5.3 Embedding Technique for Parallel Solution

We propose a novel technique that embeds a source processor array onto a target processor array having smaller number of processing elements (PE). The technique is illustrated using the pattern matching algorithm for deep packet classification. Through Markov analysis, it is proved that the performance of the target array shows the same performance as the source array. The target array is implemented and synthesized on Stratix II Altera FPGA to verify the correctness of its operation and to determine the effects of number of PEs on FPGA parameters. The synthesized target array is modeled in C language and the extensive numerical simulations are performed to verify the Markov analysis results. In addition, the throughput of the

target array is estimated and the synthesis results are discussed. These works are published in [66, 84].

1.5.4 Longest Prefix Matching

We propose a novel variable-stride multi-bit trie data structure for IP-lookup table to assist fast IP-lookup and fast routing table update. We first discuss a solution of a problem in expanding IP addresses. We give mathematical expression to determine the performance theoretically. We have done extensive numerical simulations to determine the performance of our proposed algorithm and also to determine best values for different parameters for the IP-lookup. The simulation results show that our proposed technique performs better than existing techniques in terms of lookup and update times and number of total memory access during IP-lookup. Few techniques have better time performance, although the number of memory access of their techniques are larger than that of ours. The reason of this is that they used computers with better configurations. Even though, their update times are much higher than that of ours. However, our proposed technique requires larger memory than others. But the memory requirement is quite acceptable considering the current memory price. These works are submitted in [85].

1.5.5 Best Match

We propose a novel Best Match technique required to detect best-matched English words of obfuscated spam words. We use a non-deterministic finite automaton (NFA) to build the English dictionary. We use dynamic programming with state pruning to detect the best-matched word of an obfuscated spam word in the NFA. We have done extensive numerical simulations to prove the accuracy of our proposed system. We have varied different design parameters during simulations to show the effects of these

parameters on the accuracy of the detection. Our system can detect best-matched words of the words obfuscated by spammers using five different techniques: insertion, deletion, substitution, transpose, and word boundary. Upto our knowledge, no other system can deal with all these obfuscating techniques so quickly as ours. Using simulations, we have determined the time complexity of our system which is far less than the time complexity required to simulate an NFA. We also present the time complexity analysis that agrees with the simulation results. These works are published in [86, 87].

1.6 Dissertation Organization

The rest of the dissertation is organized as follows.

In Chapter 2, we propose a string search algorithm suitable for the hardware-implementation of the deep packet classification. Through extensive numerical analyses and software simulations, we have proved that our algorithm performs better that any other string search algorithm for deep packet classification.

In Chapter 3, we propose a systematic technique for expressing the string search algorithm as a regular iterative expression to explore all possible processor arrays for DPC. Through extensive numerical analyses and software simulations, we have determined the time complexities of the processor arrays.

In Chapter 4, we propose a novel technique that embeds a source processor array onto a target processor array having smaller number of processing elements (PE). The technique is illustrated using the pattern matching algorithm for deep packet classification. We have modeled the embedded architecture in C language. Through extensive numerical analyses and software simulations, we have determined the time complexities of the processor arrays.

In Chapter 5, we propose a novel variable-stride multi-bit trie data structure

for IP-lookup table to assist fast IP-lookup and fast routing table update. We have simulated our technique using eleven publicly available routing tables to determine the performance of out proposed technique.

In Chapter 6, we propose a novel Best Match technique required to detect best-matched English words of obfuscated spam words. We have used dynamic programming (with pruning) on NFA using three different data-sets to determine the performance of our proposed technique.

Finally, we conclude our dissertation in Section 7 with future research directions.

Chapter 2

Exact Match: Sequential Solution

In this chapter, we propose a string search algorithm suitable for the hardware-implementation of the deep packet classification. This chapter is organized as follows. Section 2.1 summarizes several well-known string search algorithms with their time and space complexities. This section also discusses Boyer Moore algorithm and its shortcomings. Section 2.2 discusses our proposed modifications done in the Boyer Moore algorithm. This section also describes our algorithm with pictorial and algorithmic descriptions. Section 2.3 shows a string search example illustrating all steps of our algorithm. Section 2.4 analyzes the time complexity of our algorithm. Section 2.5 includes experimental details with their results in graphical and descriptive form. Finally, Section 2.6 concludes the chapter with a summary of our achievements.

2.1 Existing Works

String search algorithms offer improvements over the brute force algorithm that finds a P in a T in quadratic time at worst, average and best cases [88]. The improvements are mainly to reduce average and worst case time complexities. Performance of different

algorithms may vary with the inputs: T, P, and symbol parameters: n, m, and $|\Sigma|$. Table 2.1 summarizes time and space complexities of some well-known string search algorithms [88]. Almost all these algorithms have been divided into two phases: (i) preprocessing phase and (ii) searching phase. The preprocessing phase obtains preliminary results based on n, m, and $|\Sigma|$ to speedup the searching process.

Table 2.1: Summary of string search algorithms (W: Worst case; A: Average case; B: Best case).

Algorithm name		**Underlying technique**	**Complexity [88]**		
			Preprocessing phase		**Searching phase**
			Space	**Time**	**Time**
Brute force		Searches for *pattern* in every position in *text* between 0 and $(n-m)$ and shifts *pattern* by one position to the right after each attempt.	not required	not required	W: $\mathcal{O}(mn)$ A: $\mathcal{O}(mn)$ B: $\mathcal{O}(mn)$
Automata based	Automata	Based on automata theory [89].	$\mathcal{O}(m\|\Sigma\|)$	$\mathcal{O}(m\|\Sigma\|)$	A: $\mathcal{O}(n)$
	Simon	Uses deterministic finite automaton [90].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(n+m)$
Reverse	Reverse Factor	Uses the smallest suffix automaton of the reverse *pattern* [91].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(mn)$
	Turbo Reverse Factor	Uses same techniques as Reverse Factor algorithm after little modification [92] [93].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(n)$

Table 2.1: Summary of string search algorithms (W: Worst case; A: Average case; B: Best case).

Factor based	Backward Nondeterministic Dawg Matching	Uses bit-parallelism simulation of the smallest suffix automaton of Reverse Factor algorithm [94].	not available[1]	not available[1]	not available[1]
	Backward Oracle Matching	Uses suffix oracle instead of suffix automaton of *pattern* [95].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(mn)$
	Forward Dawg Matching	Uses smallest suffix algorithm of *pattern* [96].	not available[1]	not available[1]	not available[1]
Karp-Rabin		Uses hashing for preprocessing to reduce quadratic time complexity [97].	constant	$\mathcal{O}(m)$	A: $\mathcal{O}(mn)$
Shift OR		Uses bitwise techniques for preprocessing [98].	$\mathcal{O}(m + \lvert\Sigma\rvert)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	A: $\mathcal{O}(n)$
Rita		Checks last, first, and middle characters of *pattern* in order before comparing other characters [99].	$\mathcal{O}(\lvert\Sigma\rvert)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	A: $\mathcal{O}(mn)$
Galil-Seiferas		Uses a decomposition technique for preprocessing [100].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(n)$
Two Way		Divides *pattern* into two substrings for pattern matching. This algorithm requires an ordered alphabet to reduce time complexity [101].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(n)$

[1]information cannot be found in literature

Table 2.1: Summary of string search algorithms (W: Worst case; A: Average case; B: Best case).

Skip Search based	Skip Search	Uses buckets of characters in *pattern* to collect the positions of them in *pattern* [102].	$\mathcal{O}(m + \lvert\Sigma\rvert)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	A: $\mathcal{O}(mn)$
	KMP Skip Search	Uses buckets of each characters in alphabet [102].	$\mathcal{O}(m + \lvert\Sigma\rvert)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	A: $\mathcal{O}(n)$
	Alpha Skip Search	Uses buckets of positions for each factor of length $log_{\lvert\Sigma\rvert}(m)$ of *pattern* [102].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(mn)$
Morris-Pratt based	Morris-Pratt	Remembers the matched portion of *text* and *pattern* [103].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(m+n)$
	Knuth-Morris-Pratt	Improves the Morris-Pratt algorithm [104].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(m+n)$
	Colussi algorithm	Improves on the Knuth-Morris-Pratt algorithm. It divides *pattern* into two disjoint subsets. First subset is checked from left to right while second one is checked from right to left [105].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(n)$
	Galil-Giancarlo	Refines on Colussi algorithm in the searching phase [106].	$\mathcal{O}(m)$	$\mathcal{O}(m)$	A: $\mathcal{O}(n)$
	Boyer-Moore algorithm	Uses two shifts of *pattern* to speed up its searching operation. Is considered to be most efficient string search algorithm in practical cases [88] [107] [108].	$\mathcal{O}(m + \lvert\Sigma\rvert)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	A: $\mathcal{O}(mn)$
	Turbo-BM	Remembers the most recent matched substring [109].	$\mathcal{O}(m + \lvert\Sigma\rvert)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	A: $\mathcal{O}(n)$

Table 2.1: Summary of string search algorithms (W: Worst case; A: Average case; B: Best case).

Boyer-Moore based	Apostolico-Giancarlo	Remembers the most recent activities [110].	$\mathcal{O}(m + \lvert\Sigma\rvert)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	A: $\mathcal{O}(n)$
	Reverse Colussi	Divides *pattern* into two disjoint subsets [111].	$\mathcal{O}(m\lvert\Sigma\rvert)$	$\mathcal{O}(m^2)$	A: $\mathcal{O}(n)$
	Horspool	Uses one of two shifts of Boyer-Moore algorithm in a slightly different way to simplify the Boyer-Moore algorithm [112].	$\mathcal{O}(\lvert\Sigma\rvert)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	A: $\mathcal{O}(mn)$
	Quick Search	Works on one of two shifts of *pattern* like Horspool algorithm. This algorithm is good for short *pattern* and large $\lvert\Sigma\rvert$ [113].	$\mathcal{O}(\lvert\Sigma\rvert)$	$\mathcal{O}(m + \lvert\Sigma\rvert)$	A: $\mathcal{O}(mn)$
	Optimal Mismatch	Uses frequencies of characters in *pattern* [113].	$\mathcal{O}(m + \lvert\Sigma\rvert)$	$\mathcal{O}(m^2 + \lvert\Sigma\rvert)$	A: $\mathcal{O}(mn)$
	Tuned Boyer-Moore	Shifts *pattern* several times before going for the actual checking [114].	not available[2]	not available[2]	not available[2]
	Zhu-Takaoka	Uses two consecutive *text* character to determine shift [115].	$\mathcal{O}(m + \lvert\Sigma\rvert^2)$	$\mathcal{O}(m + \lvert\Sigma\rvert^2)$	A: $\mathcal{O}(mn)$
	Berry-Ravindran	Combines the concepts of Quick search and Zhu-Takaoka algorithms. It shifts *pattern* based on two consecutive *text* characters from the right side of *pattern* window [116].	$\mathcal{O}(m + \lvert\Sigma\rvert^2)$	$\mathcal{O}(m + \lvert\Sigma\rvert^2)$	A: $\mathcal{O}(mn)$

[2]information cannot be found in literature

Table 2.1: Summary of string search algorithms (W: Worst case; A: Average case; B: Best case).

	Smith	Combines Horspool and Quick Search algorithms for fast string searching [117].	$\mathcal{O}(\lvert\Sigma\rvert)$	$\mathcal{O}(m+\lvert\Sigma\rvert)$	A: $\mathcal{O}(mn)$

For long n, m, and $|\Sigma|$, Boyer-Moore algorithm is very efficient [107]. For this reason, we see many modifications of Boyer-Moore algorithm in Table 2.1. All those modifications generally aim at reducing three things: (i) implementation complexity, (ii) space requirements, and (iii) search time. To serve all these purposes, we have modified the Boyer-Moore algorithm without sacrificing its speed. Rather, we shall show that our modified algorithm is faster than Boyer-Moore algorithm in some cases.

2.1.1 Boyer-Moore Algorithm

Before describing our modifications, we first explain the original Boyer-Moore algorithm and its shortcomings for hardware implementations. The Boyer-Moore algorithm starts from the m^{th} character (y) of T and compares it with the rightmost character (x) of P. Here, two cases may occur as follows.

$y \neq x$: Here the Boyer-Moore algorithm tries to find y in other positions of P. Figure 2.1(b) shows that P is shifted right by m characters when y does not exist in P. Figure 2.1(c) shows that y exists in the left of the right end of P by d_1 characters, so in Figure 2.1(d) P is shifted right by d_1 characters.

$y = x$: Here the Boyer-Moore algorithm continues character comparison from right to left. In Figure 2.2(a), the Boyer-Moore algorithm finds a substring s in both P and T. Before s, character x of P does not match character y of T. Let Boyer-Moore algorithm finds y in the left of the position of x by d_1 characters and s in the right of the right end of P by d_2 characters. If $d_1 > d_2$, then P is

moved right by d_1 characters (Figure 2.2(b)), otherwise P is moved right by d_2 characters (Figure 2.2(c)).

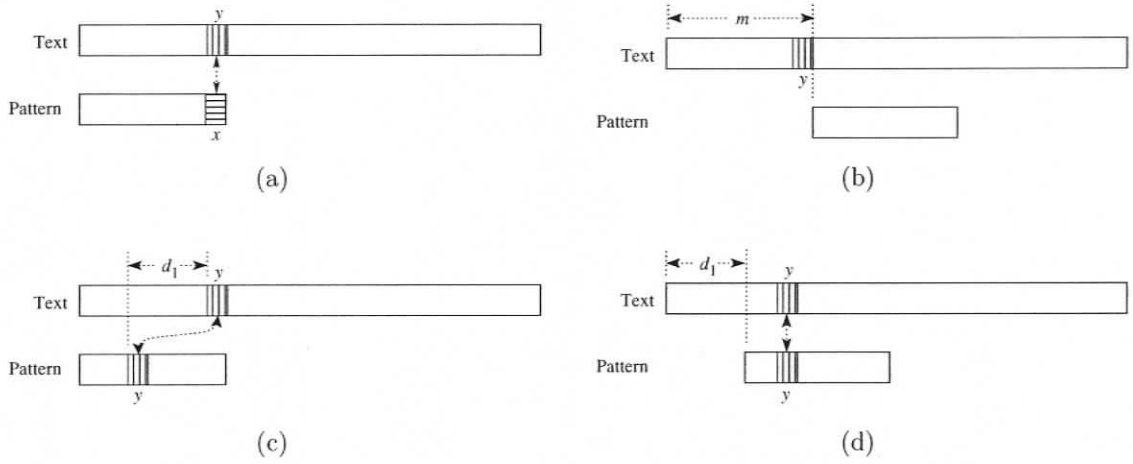


Figure 2.1: (a) The m^{th} character of T is compared with the rightmost character of P. (b) The P is shifted right by m characters as y does not exist in P. (c) y is found in the left from the right end of P by d_1 characters , so in (d), P is shifted right by d_1 characters.

From Figure 2.2(a), d_1 is the distance between two characters, whereas d_2 is the distance between two substrings. So d_1 is very easy to compute, but d_2 is more complex. To make the searching process faster, both d_1 and d_2 are preprocessed and stored for use in the searching phase. As this requires extra memory storage, we have modified the Boyer-Moore algorithm to eliminate the storage requirement and the complex d_2 calculations.

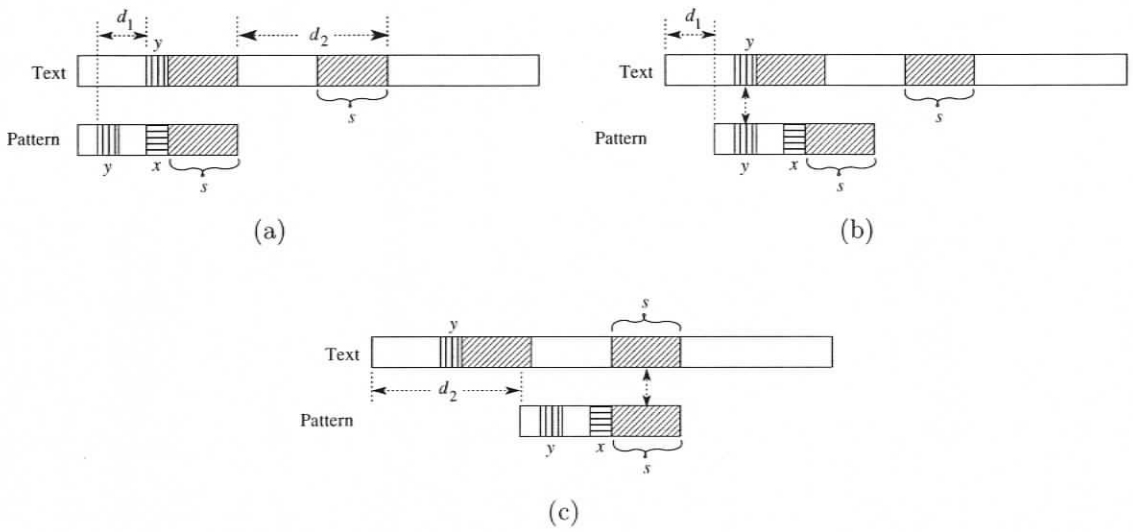


Figure 2.2: (a) after matching substring s, the Boyer-Moore algorithm calculates d_1 and d_2. (b) P is shifted right by d_1 characters if $d_1 > d_2$. (c) P is shifted right by d_2 characters if $d_1 \leq d_2$.

2.2 Proposed Algorithm

In this section, we describe the proposed modifications of Boyer-Moore algorithm followed by description of our modified algorithm.

2.2.1 Modifications of Boyer-Moore Algorithm

We showed in Section 2.1 that Boyer-Moore algorithm uses d_1, d_2, and preprocessing to speed up the searching operation. We have modified all these three steps to reduce implementation complexity and further speed up the searching operation. The proposed modifications are:

- Calculation of d_1:

    1. In Boyer-Moore algorithm, the mismatched character of T is searched in P.

In this way, the maximum value of d_1 would be m. But in our algorithm, the mismatched character of P is searched in T. Thus, d_1 can be more than m and thereby the movement of P would be larger to speed up the searching process.

2. We have added more intelligence to the calculation of d_1 by two-way checking. If the mismatched character of P is found in T, then the mismatched character of T is also compared with the corresponding character of P. This two-way checking helps more accurate and larger movements of P (i.e., larger values of d_1).

3. Assume that the distance of the mismatched character of P from the right end of P is m_1 and the mismatched character of P is at the i^{th} position of T. Assume that the distance of i^{th} character of T from the right end of T is m_2. If $m_1 > m_2$ as in Figure 2.3, then there is no possibility of finding the P in T. In that case, the searching operation would terminate. This modification speeds up the searching operation.

- Calculation of d_2: Since d_2 is very costly to compute, we have not used d_2 in our algorithm. Instead, d_1 calculations are made intelligent enough to substitute the needs for d_2.

- Preprocessing: Since our d_1 calculations are not very complex, no preprocessing is required and thereby the preprocessing space requirements are eliminated.

2.2.2 Description of Modified Boyer-Moore Algorithm

Our modified Boyer-Moore algorithm is described in Figure 2.4. This algorithm finds all the occurrences of P in T. There is one outer loop (loop1: L2–L18) and two inner loops (loop2: L5–L8 and loop3: L13–L15). The searching algorithm has to compare

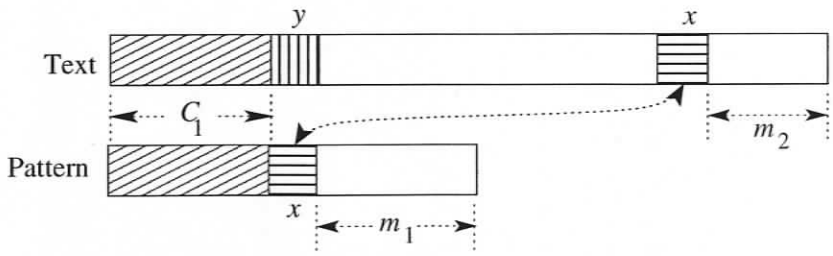


Figure 2.3: If $m_1 > m_2$, P is not in T and the algorithm terminates without any checking. If $m_1 \leq m_2$, P is shifted right to align two matched characters, x.

every n character of T for searching P in T. In our algorithm, P is shifted right by d_1 characters for checking from new position of T. So, the terminating condition of loop1 should be $n - d_1 \geq 0$. But if $n - d_1 < m$, then there is no chance of finding P in T. Thus loop1 has the condition $n - d_1 \geq m$ (L2), i.e., loop1 executes $\mathcal{O}((n - d_1)/m)$ times. Calculations of this d_1 depend on loop2 and loop3. We shall describe these calculation techniques through the following steps:

- *Step* 1: Loop2 tries to match (condition $T[d_1 + i] = P[j]$ in L5) consecutive characters (Figure 2.5). This loop executes C_1 times, i.e., consecutive C_1 characters from P and T match with each other. Here, $0 \leq C_1 \leq m$ (condition $j < m$ in L5).

- *Step* 2: Loop3 is a little bit complex. This loop executes only if loop2 terminates before m execution (i.e., P is not found in T). If P is found in T, d_1 increases by 1 to check other occurrences of P in T; otherwise, increment of d_1 is determined by loop3.

- *Step* 3: For loop3, after C_1 character match from the loop2, let the mismatched characters of P and T be x and y respectively. Loop3 first tries to find character x (condition $P[j] = T[d_1 + i]$ in L15) in the range from $(d_1 + C_1 + 1)$ to $(n + C_1 + 1 - m)$ of the string T (condition $d_1 + i = n + j + 1 - m$ in L15).

  ⋆ *Step* 3(i): If x is found after $(n + C_1 + 1 - m)$, then $(m - C_1 - 1) > (n - d_1 -$

procedure $Modified\_Boyer\_Moore(T, n, P, m)$

$//T[0 : n-1]$ is Text

$//P[0 : m-1]$ is Pattern

$L1:$ $d_1 \leftarrow 0$ $//d_1 =$ Starting position of T, from which P is to be searched

$L2:$ $loop_1$: **while** $(n - d_1) \geq m$

$L3:$ $i \leftarrow 0$ $//i =$ index of T

$L4:$ $j \leftarrow 0$ $//j =$ index of P

$L5:$ $loop_2$: **while** $j < m$ **AND** $T[d_1 + i] = P[j]$

$L6:$ $i \leftarrow i + 1$

$L7:$ $j \leftarrow j + 1$

$L8:$ **end** $loop2$

$L9:$ **if** $j = m$

$L10:$ ' Match found at d_1'

$L11:$ $d_1 \leftarrow d_1 + 1$

$L12:$ **else**

$L13:$ $loop3$: **repeat**

$L14:$ $i \leftarrow i + 1$

$L15:$ **until** $d_1 + i = n + j + 1 - m$ **OR** $(P[j] = T[d_1 + i]$ **AND**
$((2 \times j - i) < 0$ **OR** $T[d_1 + j] = P[2 \times j - i]))$

$L16:$ $d_1 \leftarrow d_1 + i - j$

$L17:$ **endif**

$L18:$ **end** $loop1$

end $Modified\_Boyer\_Moore$

Figure 2.4: Modified Boyer-Moore Algorithm

$C_1 - C_2 - 1$), this means there is no chance for P to be found in T. If the loop3 does not find x within that range, then the program terminates without any other

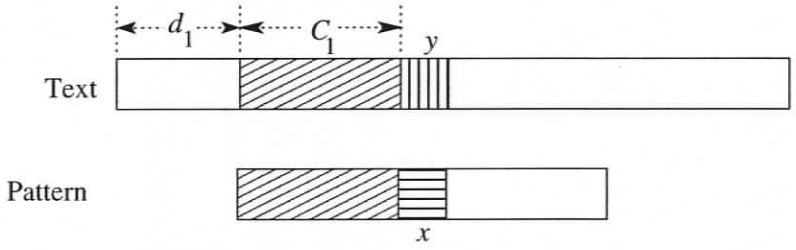


Figure 2.5: loop (L5–L8) tries to match characters from T and P. After matching C_1 characters, y of T does not match with x of P.

checking.

⋆ *Step* 3(ii): Figure 2.6(a) shows that the loop3 finds x, C_2 characters right within the range from $(d_1 + C_1 + 1)$ to $(n + C_1 + 1 - m)$. Now, if $C_2 > C_1$, then there is no need to check y in P. So in Figure 2.6(b), d_1 is incremented by C_2. Clearly, $0 \leq C_2 \leq (n - m - d_1)$.

⋆ *Step* 3(iii): In Figure 2.6(c), since $C_2 \leq C_1$ (condition $(2 \times j - i) < 0$ in L15), character y is checked with the character in the $(C_1 - C_2)^{\text{th}}$ position of P (condition $T[d_1 + j] = P[2 \times j - i]$ in L15) in Figure 2.6(d).

⋆ *Step* 3(iv): If $(C_1 - C_2)^{\text{th}}$ character of P is also y, then Figure 2.6(e) shows that d_1 is incremented by C_2.

⋆ *Step* 3(v): If $(C_1 - C_2)^{\text{th}}$ character of P is not y, then the loop3 continues to execute to find another match of character x in T.

Through these steps, our algorithm calculates the shift of P for fast string search. In the next section, we shall give a comprehensive illustrative example of a string search based on our algorithm.

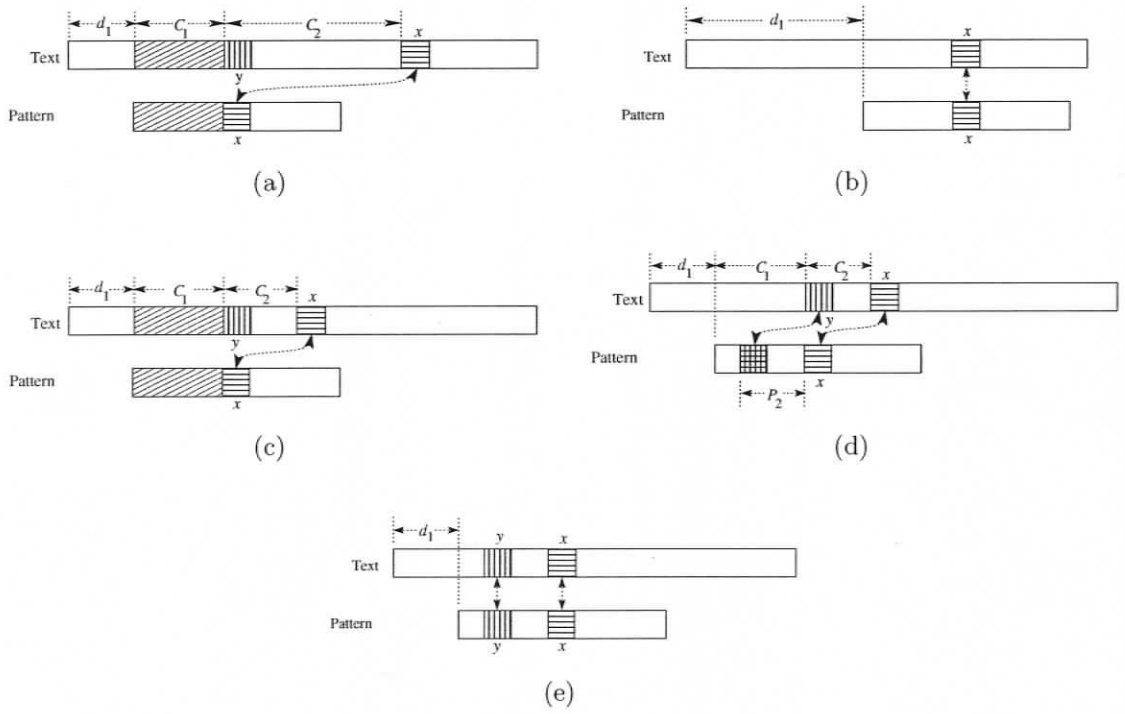


Figure 2.6: How the loop3 of Figure 2.4 works. In (a), mismatched character, x of P is found in T, C_2 characters right. Since $C_1 < C_2$, P is shifted right by C_2 characters in (b). In (c), mismatched character, x of P is found in T, C_2 characters right. Since $C_1 > C_2$, in (d), mismatched character y of T is searched in P. As y is found in P, P is shifted right by C_2 characters in (e).

2.3 An Illustrative Example

Here, we give an example showing all concepts of our algorithm in Figure 2.7. We are trying to find P ('AB_8CW') in T ('NMOAB_9A8Z0^CABAB_8CWAN'). Figure 2.7(a) shows that our algorithm tries to find character matching from index 0 (*Step* 1). But first character from T (Text[0]='N') does not match that of P (Pattern[0]='A'). So, $C_1 = 0$ here. Then, according to *Step* 3, we step forward to find

the mismatched character of P (Pattern[0]='A') in T. Figure 2.7(b) shows that 'A' is found at Text[3]. Thus $C_2 = 3$. Here, $C_1 < C_2$. *Step* 3(ii) implies to shift P right by $C_2 = 3$ characters in Figure 2.7(c). So, $d_1 = 3$. Figure 2.7(c) indicates that we have to find now character matching from index 4 according to *Step* 1. After matching three characters, fourth character from T (Text[d_1+3]='9') does not match with that of P (Pattern[3]='8'). Again *Step* 3 comes into play and finds the mismatched character of P (Pattern[3]='8') in T. '8' is found in Text[8] (Figure 2.7(d)). Here, $(C_1 = 3) > (C_2 = 2)$. So, *Step* 3(iii) indicates to compare mismatched character of T ('9') with $(C_1 - C_2) = 1^{st}$ character of P ('B'). Since those two characters are unequal, we have to search the mismatched character of P ('8') again in T for *Step* 3(v). There is another '8' in Text[18] as in Figure 2.7(e). Here, since $(C_2 = 12) > (C_1 = 3)$, *Step* 3(ii) moves P forward by $C_2 = 12$ characters. So, d_1 becomes $d_1 + C_2 = 15$. *Step* 1 indicates to compare characters of P (from Text[d_1+0]) and T (from Pattern[0]). Figure 2.7(f) shows that P is found at $d_1 = 15$ of T. After this finding, we go for more findings of P in T. According to *Step* 2, P is moved forward by one character (Figure 2.7(g)). So, now d_1 becomes $d_1 + 1 = 16$. Again after applying *Step* 1 and *Step* 3, mismatched character of P ('A') is found at index 21 as shown in Figure 2.7(h). But from *Step* 3(i), we can conclude that since $m_1 > m_2$, we can safely terminate our searching without further checking.

From Figure 2.7, we can easily calculate the number of total character comparisons. Table 2.2 shows the number of character comparisons in the given example. Here total number of character comparisons is 33 for $n = 23$. Considering the number of character comparisons as an indication of the execution time, we can say that the time requirement in this particular example is 33, which is less than the maximum time required for our algorithm ($2 \times n = 2 \times 23$), as will be shown in Section 2.5.

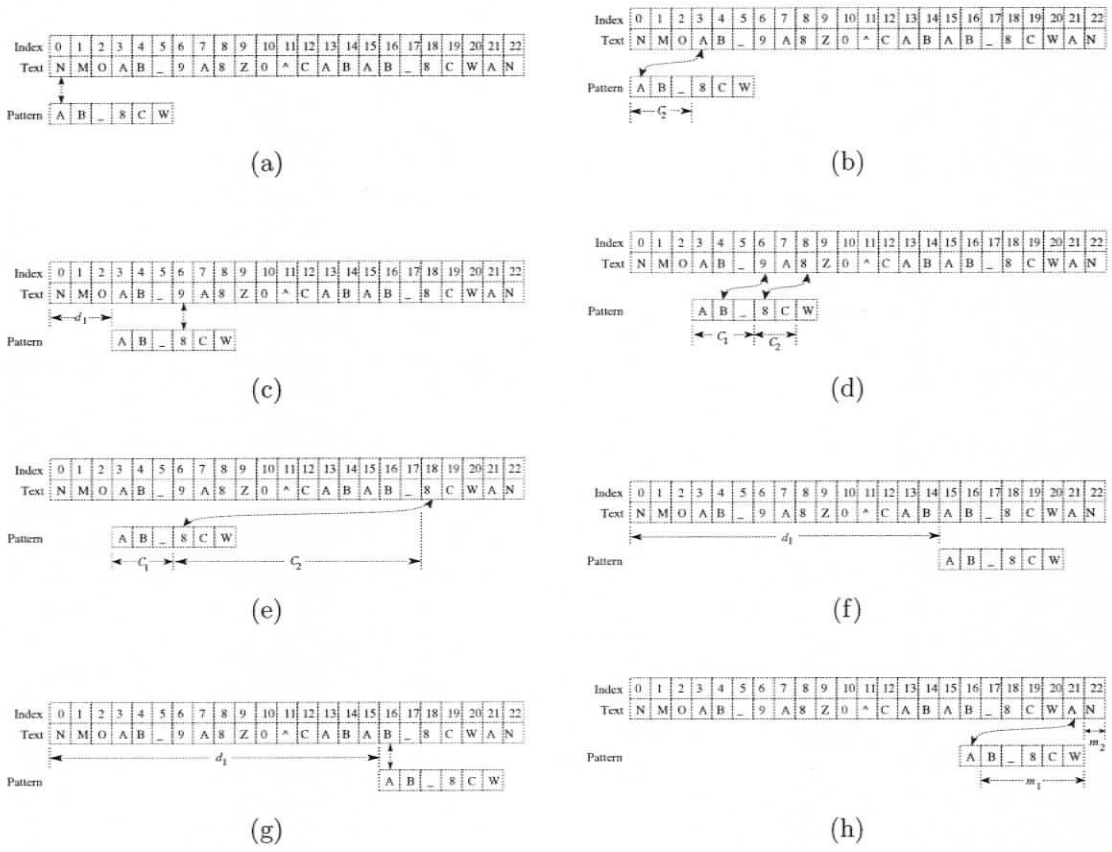


Figure 2.7: An example of our modified Boyer-Moore algorithm. The number of character comparisons are: one in (a), three in (b), four in (c), three in (d), ten in (e), six in (f), one in (g), and five in (h).

2.4 Time Complexity Analysis

There is no preprocessing in our algorithm and no preprocessing space is required. So, we analyze only the $\mathcal{O}$ time complexity for our algorithm to show its speed. There are three main loops in our proposed algorithm as shown in Figure 2.4. Almost all the statements are within loop1, with two inner loops (loop2 and loop3) forming its main body. There are four statements outside these two inner loops. Execution time

Table 2.2: Number of character comparisons in the given example.

Figure index	# of Comparisons
Figure 2.7(a)	1
Figure 2.7(b)	2
Figure 2.7(c)	4
Figure 2.7(d)	3
Figure 2.7(e)	10
Figure 2.7(f)	6
Figure 2.7(g)	1
Figure 2.7(h)	5
All	33

of these two inner loops would be much more than these four statements. So, for ease of calculations, we consider only these loops for $\mathcal{O}$ complexity analysis. Also, we assume that the two inner loops have same execution time. We have done the complexity analysis of our algorithm for worst, best, and average cases.

2.4.1 Worst Case

When $d_1 = 1$ and $m = 1$, loop1 can run maximum $\mathcal{O}(n)$ times. Since $d_1 = 1$, the loop3 would execute $\mathcal{O}(1)$ times. Again, another loop2 may run maximum $\mathcal{O}(m)$ times. Thus, the worst case time complexity, T_w is $\mathcal{O}(n(m+1)) = \mathcal{O}(nm)$.

We can do this analysis in another way. If loop3 executes maximum $\mathcal{O}(n)$ times, then loop1 would execute only once. Then, the complexity is $\mathcal{O}(1(n+m))$ $= \mathcal{O}(n+m)$, which is less than $\mathcal{O}(nm)$. So,

$$T_w = \mathcal{O}(nm)$$

2.4.2 Best Case

In the best case, loop2 executes $\mathcal{O}(1)$ times, but loop3 never executes. So, $d_1 = 0$. Then, loop1 executes $\mathcal{O}(n/m)$ times. Thus, the best case time complexity, T_b is $\mathcal{O}(1 \times (n/m)) = \mathcal{O}(n/m)$.

2.4.3 Average Case

For the average case, let C_1 be the average consecutive number of characters of P and T that match each other and loop3 executes C_2 times on average. This means, loop2 executes $\mathcal{O}(C_1)$ times and loop1 executes $\mathcal{O}((n - d_1)/m)$ times, where a is incremented by C_2 in each iteration. Let the loop1 executes z times. Then,

$$\frac{n - z \times C_2}{m} = z$$

Therefore,

$$z = \frac{n}{C_2 + m}$$

Thus, the average case time complexity, T_a can be written as

$$T_a = \mathcal{O}((n - n \times C_2/(C_2 + m)) \times (C_1 + C_2)/m)$$

Here, using summation limits from Section 2.2,

$$C_1 = \sum_{i=0}^{m} i \times prob_1$$

$$C_2 = \sum_{i=0}^{n-m-d_1} i \times prob_2$$

where $prob_1$ is the probability that substring of i characters of P and T match with each other and $prob_2$ is the probability that consecutive i characters of P and T mismatch all together and two characters (before and after the mismatch substring) of P and T match with each other. So $prob_1$ can be written as,

$$prob_1 = p_{mm} \times p_m^i$$

where p_m = match probability = $1/|\Sigma|$, and p_{mm} = mismatch probability $1 - r = 1 - 1/|\Sigma|$. And $prob_2$ can be written as,

$$prob_2 = p_m \times p_{mm}^i$$

where, $p_m = 1/|\Sigma|^2$ and $p_{mm} = 1 - 1/|\Sigma|^2$.

Therefore, (details are in [55])

$$C_1 = \sum_{i=1}^{m} i \times \left(1 - \frac{1}{|\Sigma|}\right) \times \left(\frac{1}{|\Sigma|}\right)^i \tag{2.1}$$

$$\approx \frac{1}{|\Sigma| - 1} \tag{2.2}$$

$$C_2 = \sum_{i=0}^{n-m-d_1} i \times \frac{1}{|\Sigma|^2} \times \left(1 - \frac{1}{|\Sigma|^2}\right)^i \tag{2.3}$$

$$= m + d_1 - n \tag{2.4}$$

$$T_a = \mathcal{O}(n) \tag{2.5}$$

The results of these analyses are tabulated in Table 2.3.

Table 2.3: Complexities of proposed modified Boyer-Moore algorithm after numerical complexity analysis.

		Modified Boyer-Moore algorithm
Time requirement	**Worst**	$\mathcal{O}(mn)$
	Average	$\mathcal{O}(n)$
	Best	$\mathcal{O}(n/m)$

2.5 Experimental Results

We took four schemes of $|\Sigma|$: (i) $|\Sigma|= 2$ for searching binary data, (ii) $|\Sigma|= 26$ for searching character strings, (iii) $|\Sigma|= 127$ for searching 7-bit ASCII, and (iv) $|\Sigma|= 65$

535 for searching 16-bit parallel data. We ran our algorithm (Figure 2.4) 10^5 times for each of these four values of $|\Sigma|$ to get data for experiments. Each time, we have randomly generated each character of P and T and their lengths (m, n) of them for a given $|\Sigma|$. We chose a random function that generated numbers uniformly spread over a certain range so that we can get random samples in each experiment. For each iteration, we collected the following parameters: a, b, c, m, and n, where, a is total number of execution of the loop1, b is total number of execution of the loop2, c is total number of execution of the loop3, and $1 \leq n, m \leq 1000$. With these data, we did following experiments:

2.5.1 Determining Parameters for Run Time Calculation

In Section 2.4, we considered only the two loops (loop2 and loop3) of Figure 2.4 for complexity analysis. In this section, we prove this consideration practically.

In Figure 2.4, loop2 and loop3 are inside loop1. Loop1 also has four more statements (L6, L7, L9, and L11 or L16). Loop2 has two statements and Loop3 has one statement. So, there are a total of three statements in these two loops. In addition, each loop must execute at least one comparison in each iteration. Thus there are total five statements to be executed per iteration of two inner loops. For ease of calculation, we consider that all types of statements take same time for execution.

We did some experiments to prove that execution time of two inner loops (loop2 and loop3) is much bigger than these four statements. Assume Δ is the difference between the execution times of two inner loops and four statements outside the two inner loops. Then,

$$\Delta = 5 \times (b + c) - 4 \times a$$

We plotted Δ versus sample index in Figure 2.8 and Figure 2.9 to prove that we can take only two inner loops for time complexity analysis of our algorithm. Figure 2.9

represents an enlarged window of Figure 2.8 for 500 sample data to get clear view of the plots.

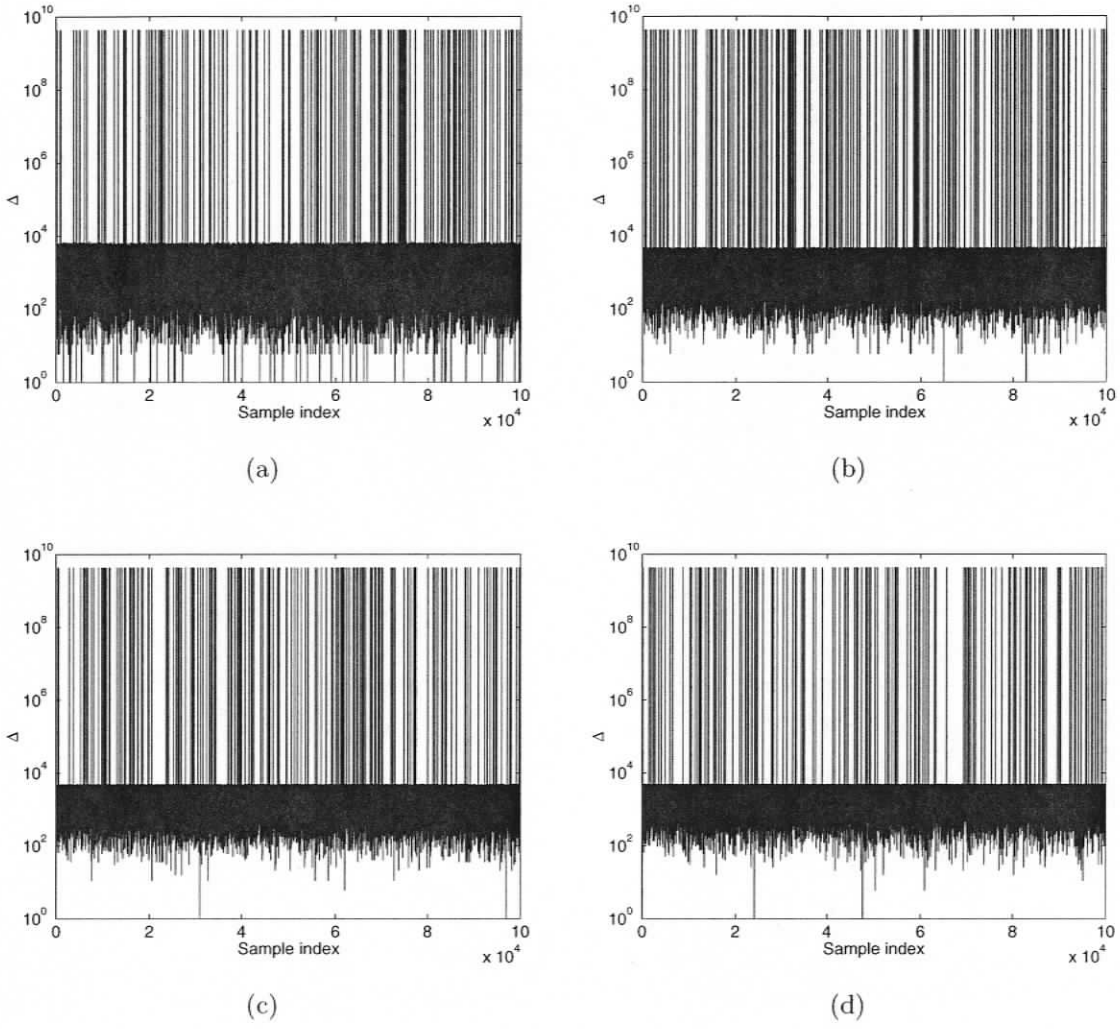


Figure 2.8: Plot for Δ vs. sample index, where Δ is the difference of execution times between two inner loops (loop2 and loop3) and four statements in L6, L7, L9, and L11 or L16 of Figure 2.4. The plot in (a) is for $|\Sigma| = 2$, (b) is for $|\Sigma| = 26$, (c) is for $|\Sigma| = 127$, and (d) is for $|\Sigma| = 65\ 535$.

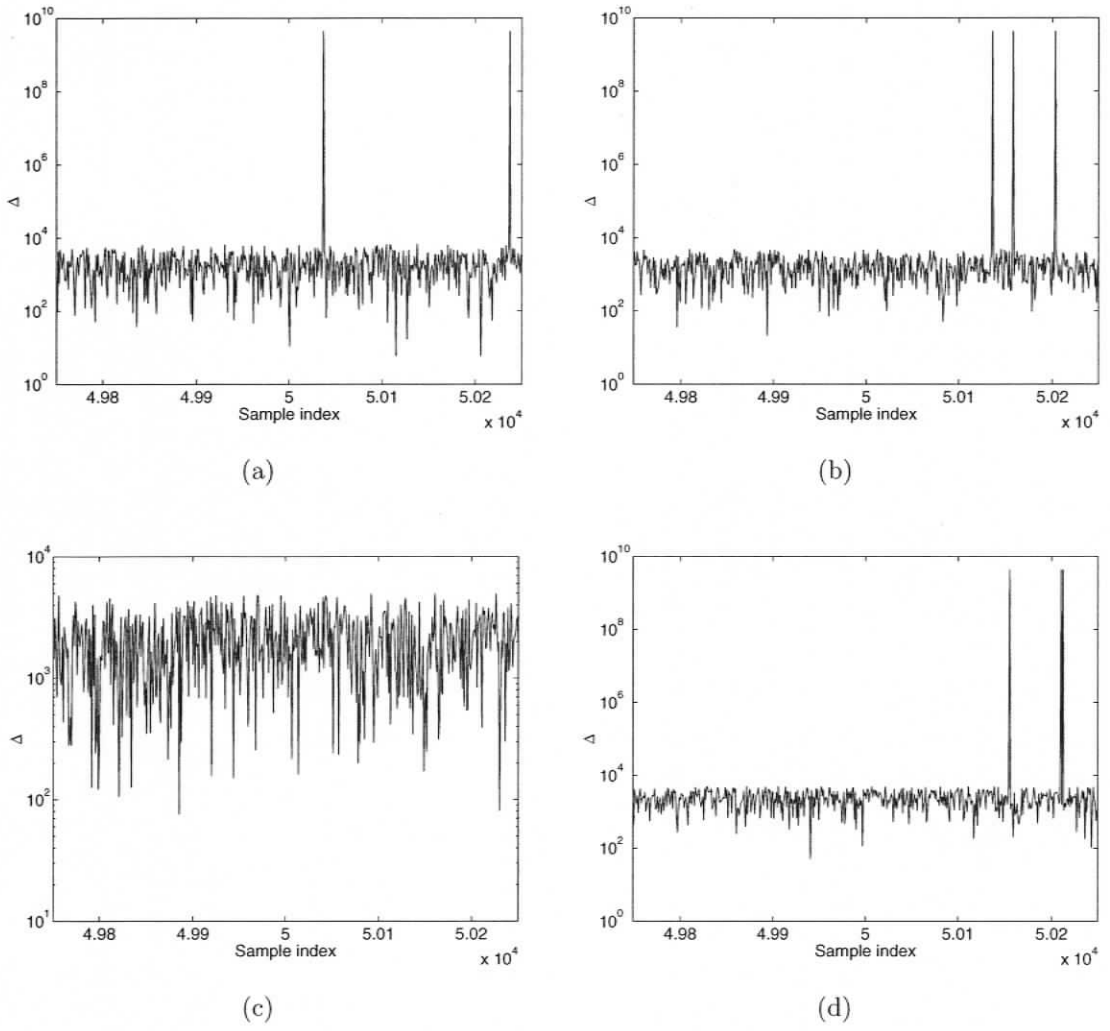


Figure 2.9: Redrawing of plots in Figure 2.8 for 500 sample data. The plot in (a) is for $|\Sigma| = 2$, (b) is for $|\Sigma| = 26$, (c) is for $|\Sigma| = 127$, and (d) is for $|\Sigma| = 65\,535$.

Both Figure 2.8 and Figure 2.9 show that average value of Δ is an order of 10^3 in all values of $|\Sigma|$ although Δ can vary with T, P, n, and m. Even for lower values of $|\Sigma|$, small number of samples have $\Delta < 10^2$ as shown in Figure 2.8(a) ($|\Sigma|= 2$), Figure 2.9(a) ($|\Sigma|= 2$), Figure 2.8(b) ($|\Sigma|= 26$), and Figure 2.9(b) ($|\Sigma|= 26$). While

the number of samples having $\Delta < 10^2$ becomes insignificant as $|\Sigma|$ increases in Figure 2.8(c) ($|\Sigma|$= 127), Figure 2.9(c) ($|\Sigma|$= 127), Figure 2.8(d) ($|\Sigma|$= 65 535), and Figure 2.9(d) ($|\Sigma|$= 65 535).

From these experiments, we can say that run time of our algorithm is an order of $(b+c)$ i.e. only the two inner loops (L5–L8and L13–L15) can be taken for complexity analysis of Figure 2.4. This value is indeed the number of character comparisons by our algorithm for a given T and P. Thus the time complexity analyses also give the number of character comparisons by our algorithm.

2.5.2 Time Complexity Calculation

Based on run times of our algorithm, time complexities are determined. According to Section 2.5.1, we calculated $(b+c)$ as run time for the four different values of $|\Sigma|$. We normalized these values by n to get a meaningful result in each experiment.

Figure 2.10 is the result of experiments for $|\Sigma| = 2$. The plot of the experimental results in Figure 2.10(a) and Figure 2.10(b) show that normalized complexity never exceeds 2. The histogram in Figure 2.10(c) also proves this observation. So, we can say that worst case complexity is $\mathcal{O}(2n)$ here. From Figure 2.10, we can also say that average run time is near $0.9n$, i.e., average time complexity is $\mathcal{O}(n)$ here.

Figure 2.11 shows normalized run time versus sample index, when $|\Sigma| = 26$. In Figure 2.11(a) and Figure 2.11(b), the normalized run time is just over 1. This observation is verified by the histogram in Figure 2.11(c). Here average run time is estimated as $0.6n$. Thus both worst case and average case time complexity are $\mathcal{O}(n)$ here.

Figure 2.12 is for the experimental results with $|\Sigma| = 127$. In Figure 2.12(a) and Figure 2.12(b), normalized run time never exceed 1. Figure 2.12(c) shows that average normalized run time is 1. So, worst case and average case complexities both

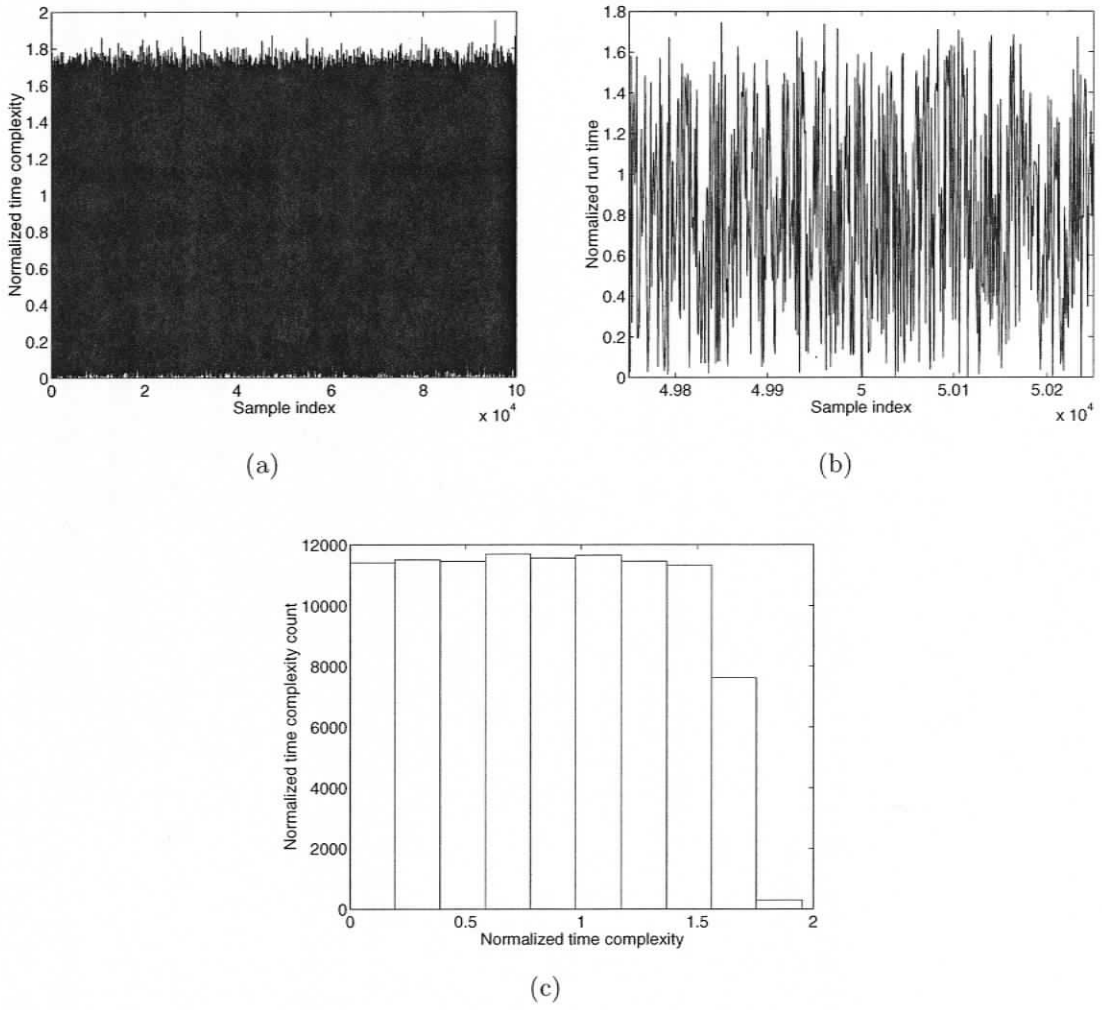


Figure 2.10: For $|\Sigma| = 2$, experimental results are plotted in (a) for all data and in (b) for 500 data, and the histogram is in (c)

are $\mathcal{O}(n)$.

Figure 2.13(a) and Figure 2.13(b) show the normalized search time versus sample index for $|\Sigma|= 65\ 535$. Here, sample data show the same behavior as the data for $|\Sigma|= 127$. But Figure 2.13(c) shows that more experimental results' normalized run time is near 1 in this case. Here both average and worst case time complexities are

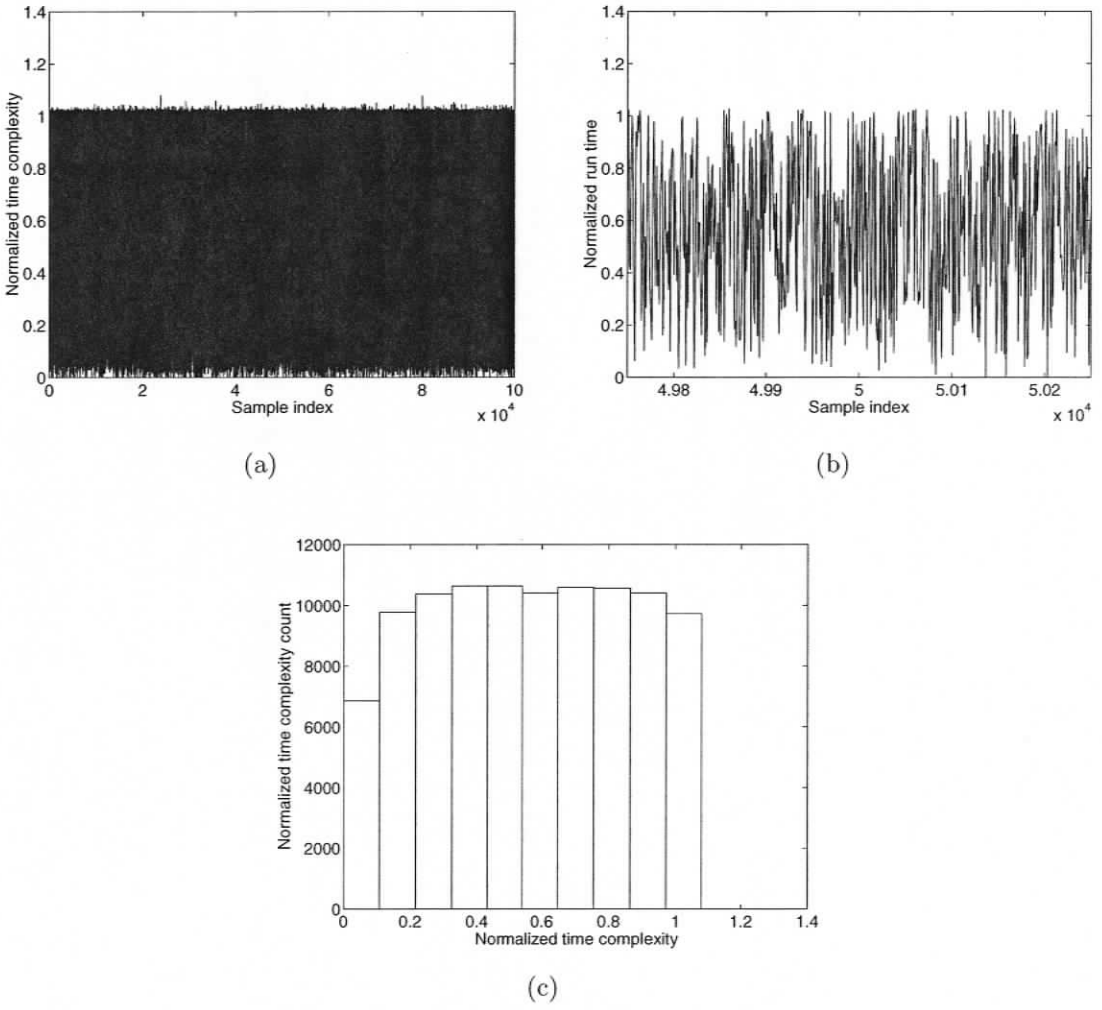


Figure 2.11: For $|\Sigma| = 26$, experimental results are plotted in (a) for all data and in (b) for 500 data, and the histogram is in (c)

also $\mathcal{O}(n)$.

We can conclude the following facts of our algorithm from the above experiments:

1. Normalized run time never exceeds 2. So, worst case time complexity is $\mathcal{O}(2n) = \mathcal{O}(n)$ [118] instead of $\mathcal{O}(nm)$ as in our theoretical analysis. As the value of

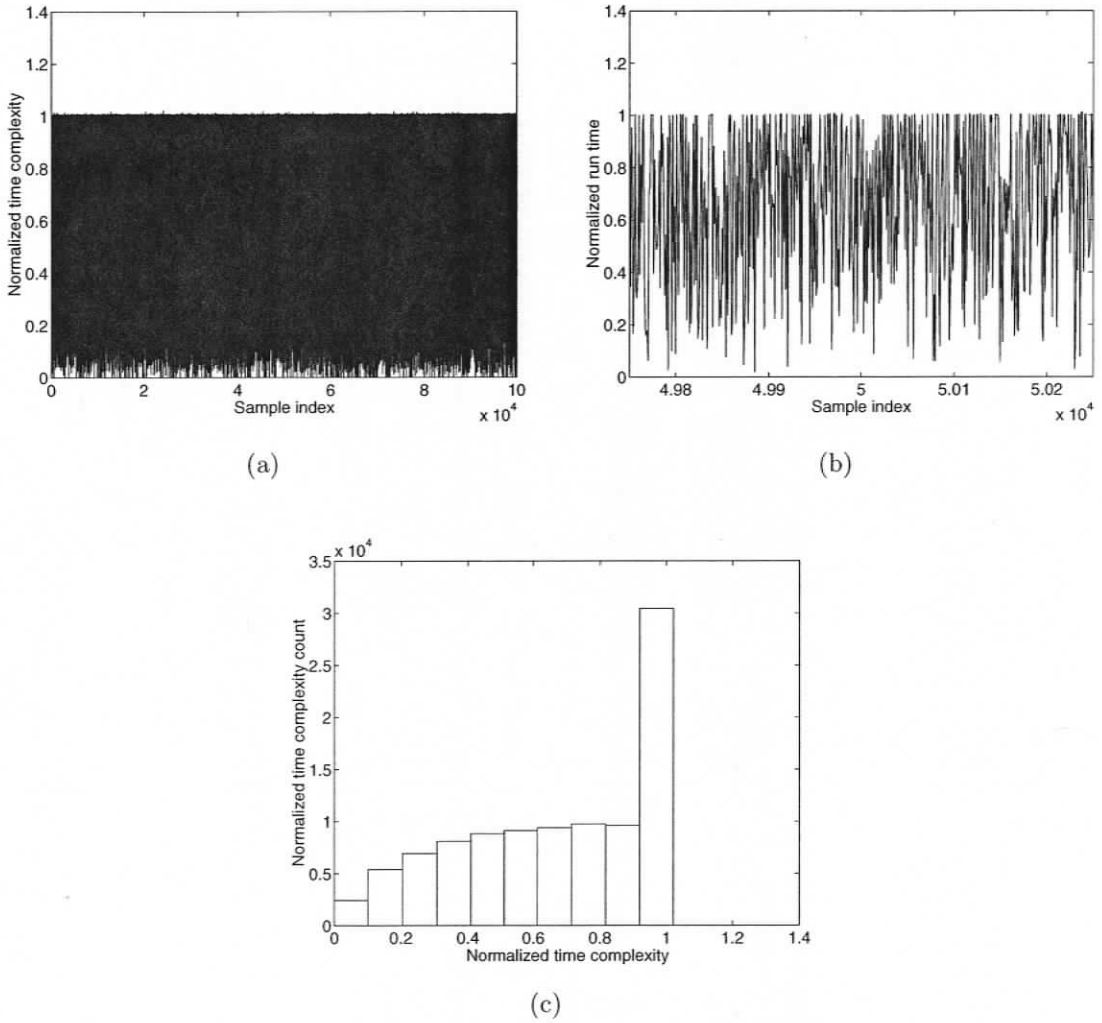


Figure 2.12: For $|\Sigma| = 127$, experimental results are plotted in (a) for all data and in (b) for 500 data, and the histogram is in (c)

$|\Sigma|$ increases, normalized run time is restricted to 1. This means the worst case time complexity is $\mathcal{O}(n)$. To make it more clear, we have plotted maximum normalized run time complexity vs. $|\Sigma|$ in Figure 2.14. From the graph, T_w is $\mathcal{O}(1.95n)$, i.e. $\mathcal{O}(n)$

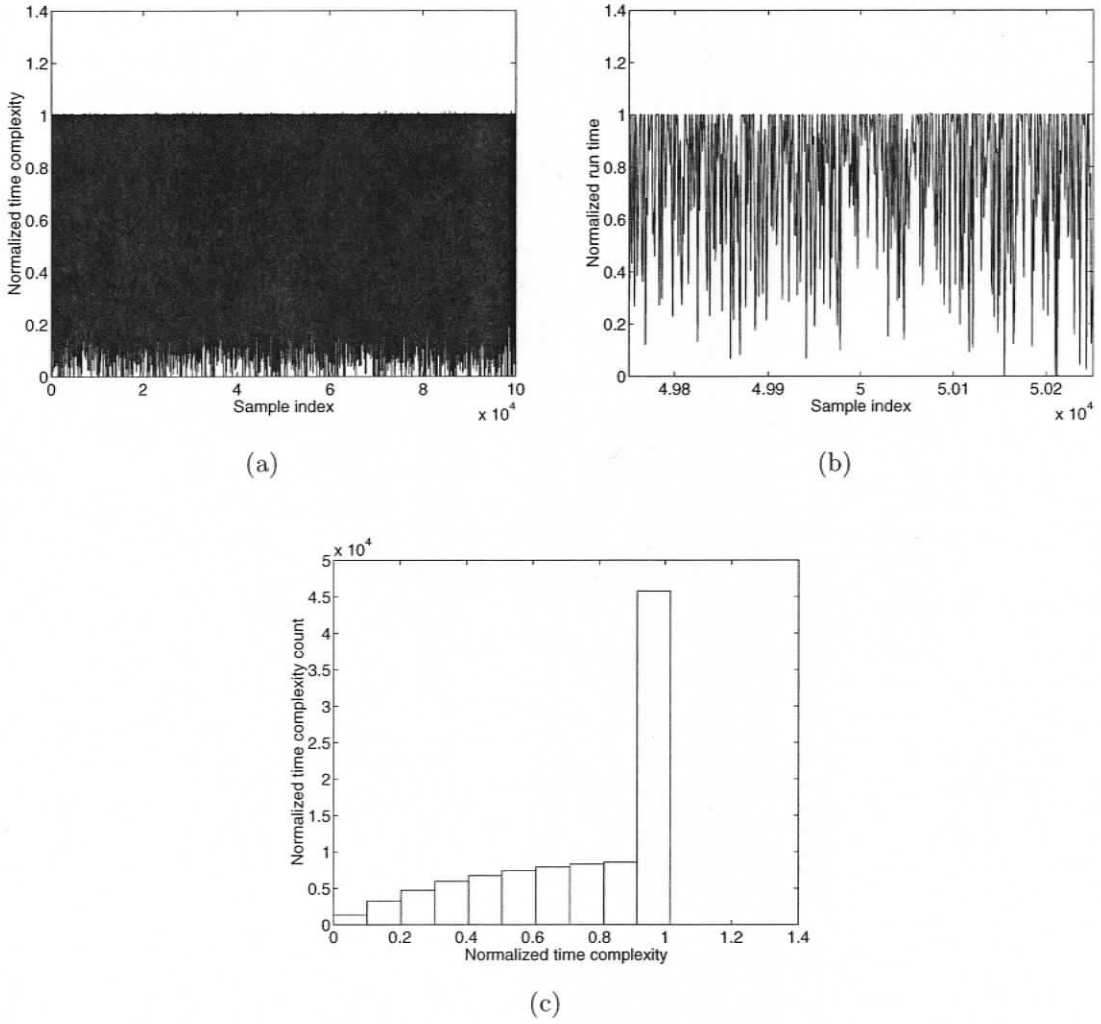


Figure 2.13: For $|\Sigma| = 2^{16} - 1 = 65535$, experimental results are plotted in (a) for all data and in (b) for 500 data, and the histogram is in (c)

Thus our algorithm performs better for larger $|\Sigma|$.

2. Average normalized run time is less than 1 for lower values of $|\Sigma|$. But as the value of $|\Sigma|$ increases, average normalized run time also increases up to 1. For larger values of $|\Sigma|$, average normalized run time is clearly 1 from his-

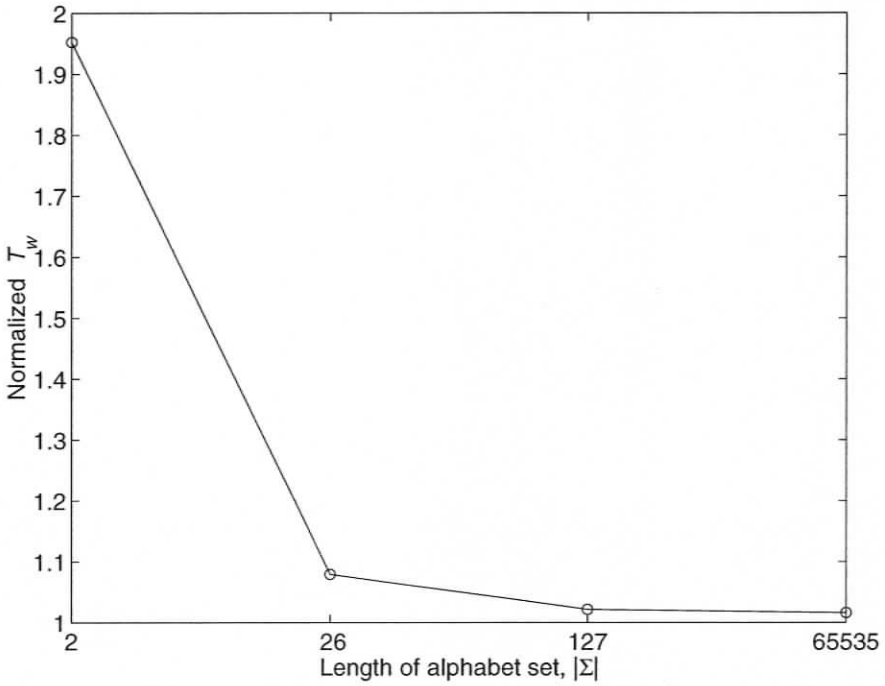


Figure 2.14: Effect of $|\Sigma|$ on search performance.

tograms (Figure 2.11(c), Figure 2.12(c), Figure 2.13(c)). Thus the average case time complexity in $\mathcal{O}(n)$ as in our theoretical analysis.

From these experiments, we got time complexities different from theoretical time complexities that we got in Section 2.4. We tabulate both theoretical and experimental time complexities in Table 2.4.

2.5.3 Effect of m on Run Time

We did some experiments to see the effects of m on run time. For these experiments, we divided normalized run time into five groups of m: (i) $1 \leq m \leq 10$, (ii) $10 < m \leq 50$, (iii) $50 < m \leq 200$, (iv) $200 < m \leq 500$, and (v) $500 < m \leq 1000$. Then, we calculated median of the normalized run time of each group and plotted the results

Table 2.4: Complexities of proposed modified Boyer-Moore algorithm after experiments.

		Modified Boyer-Moore algorithm	
		Theoretical	**Experimental**
Time requirement	**Worst**	$\mathcal{O}(mn)$	$\mathcal{O}(n)$
	Average	$\mathcal{O}(n)$	$\mathcal{O}(n)$
	Best	$\mathcal{O}(n/m)$	$\mathcal{O}(n/m)$

in Figure 2.15. Median is taken, since it gives better measure on skewed data.

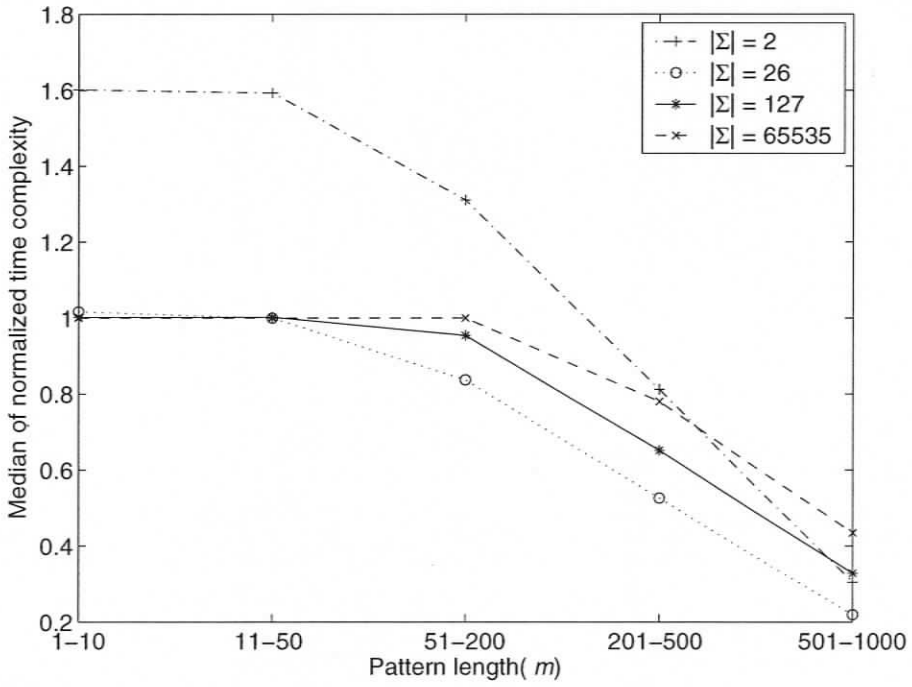


Figure 2.15: Effect of P length on search performance.

In Figure 2.15, time complexities become lower for higher value of m in every

case. These experiments prove that our algorithm works better for large m.

2.5.4 Effect of n on Run Time

Finally, we did similar experiments as in Section 2.5.3 to see the effects of n on run time. Here five groups of n are: (i) $1 \leq n \leq 50$, (ii) $50 < n \leq 150$, (iii) $150 < n \leq 300$, (iv) $300 < n \leq 600$, and (v) $600 < n \leq 1000$. Figure 2.16 shows the graph of median of the normalized run time versus n.

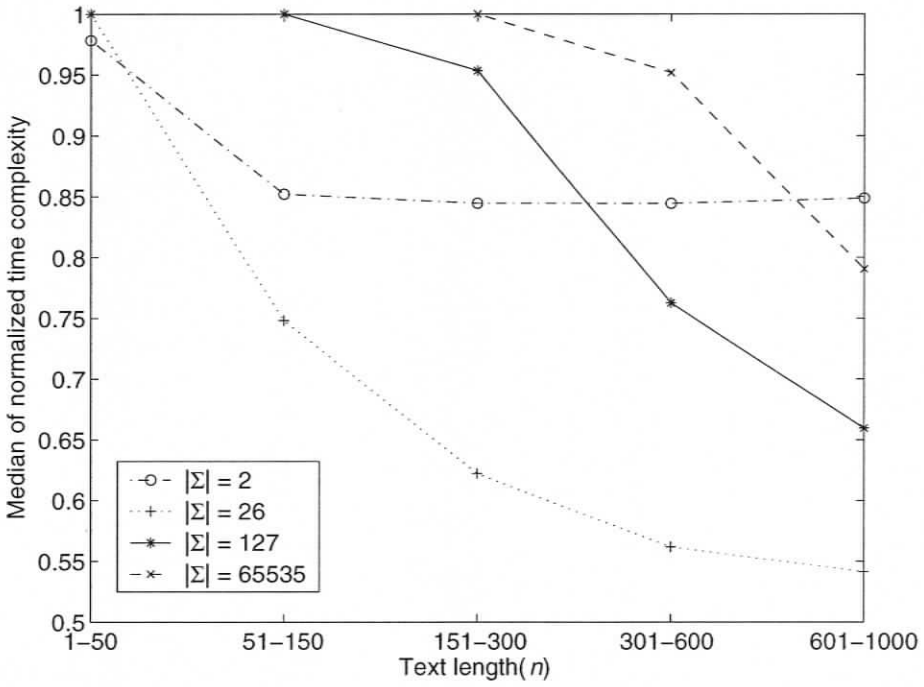


Figure 2.16: Effect of T length on search performance.

In Figure 2.16, time complexities become lower for higher value of n in every case as in Figure 2.15. These experiments prove that our algorithm also works better for large n.

In the section, we have calculated and proved some statements through different

experiments. In Section 2.5.2, We have calculated time complexities of our algorithm. Experiments in Section 2.5.2 have also proved that our algorithm works better for larger values of $|\Sigma|$. Our algorithm also works better for larger values of m and n as proved in Section 2.5.3 and Section 2.5.4.

2.6 Conclusion

For the hardware designing of the deep packet classification unit, our aim is to minimize implementation complexity and memory requirements, and to speed up the searching operation. We have minimized the implementation complexity of Boyer-Moore algorithm and memory requirements as there is no preprocessing and at the same time retain the speed of original Boyer-Moore algorithm. Through complexity analysis and experiments, we have shown that our algorithm complies with the time complexity of Boyer-Moore algorithm and even surpasses the speed of Boyer-Moore algorithm in some cases.

We have proved in Section 2.5 that our algorithm's performance increases with the increase of n, m, and $|\Sigma|$ as needed by deep packet classification. Again after comparison between Table 2.1 and Table 2.4, we can conclude that our algorithm is superior to all other string matching algorithms not only in hardware implementations but also for any string searching operation with respect to both time and space complexities.

Chapter 3

Exact Match: Parallel Solution

This chapter presents a systematic methodology for obtaining several processor array architectures for deep packet classification. A hardware implementation for the algorithmic search engine for packet classification can be assumed to have the following characteristics:

- 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 chapter is organized as follows: Section 3.1 discusses the literature related to parallel algorithms and hardwares for the string search problem. Section 3.2 introduces the systematic methodology we employed to design the processor array architecture. Sections 3.3, 3.4, and 3.5 describe the resulting processor arrays derived in Section 3.2. Section 3.6 discusses the complexity analyses of our proposed hardwares. We verify

the analysis results of the time complexity in Section 3.7 by extensive numerical simulations. In Section 3.8, we compare our design with previously designed processor arrays for the string search algorithm. Finally, we conclude our chapter in Section 3.9.

3.1 Existing 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.

3.1.1 Parallel String Search Algorithms

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

J. Jájá has proposed a parallel algorithm for string searching in [119]. 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 [120, 121], 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 [122], Galil also has designed a CRCW-PRAM constant-time optimal parallel algorithm.

In [123], Misra uses the theory of powerlists [124] to develop the 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 [125], 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 [126]. 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.

3.1.2 Parallel Hardwares for String Searching

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

Takefuji, Tanaka, and Lee, in [127], 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 [128] 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 add-hoc implementation technique that needs verifications after implementation.

Isenman and Shasha [129] developed hardware for string matching using a deterministic finite state automaton based on the standard technique of Knuth-Morris-Pratt algorithm [130]. The hardware consisted of an AT&T 32100 microprocessor that implemented the compiled code for the UNIX System command `fgrep`. The controller used 28 single character comparators together with four 16 bit adders. The speed of the system depended on the complexity of the query but the use of multiple comparators in parallel enabled 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.

3.1.3 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 [131] 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 (a) require few operations to be implemented using few simple cells; (b) local and regular data and control flow requirements; (c) inherent pipelining and multiprocessing. 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

pre-assumptions 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% 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 [132] devised a processor array to compare two strings based on the longest common subsequence 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 [133] 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 processor array for the string search problem in [134] that required preprocessing and search phases. The algorithm for the preprocessing phase was expressed as an 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 [135], Michailidis and Margaritis solved the same as in [134] using non-deterministic finite automata. In the same way as in [134], they used dependency graph. Their approach has the same complexities and problems as in [134].

Sastry, Ranganathan, and Remedios [136] 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

$$\left(n + \left\lceil \frac{N}{2} \right\rceil\right) \times 25 \times 10^{-9} s \tag{3.1}$$

where N is the number of processing elements. Equation 3.1 assumes that the processing can be completed in a single pass. If multiple passes are required, the required time is

$$\left((m-1) \times 2 \times \left\lceil \frac{N}{2} \right\rceil + n + \left\lceil \frac{N}{2} \right\rceil\right) \times 25 \times 10^{-9} s \tag{3.2}$$

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

3.2 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 [58–60, 64]. However most of these techniques were only able to deal with two-dimensional (2-D) 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 4-D dependence graph. The first author proposed a formal algebraic procedure for processor array implementation starting from a regular iterative algorithm with arbitrary dimensions [64]. 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 subsections.

3.2.1 Expressing the algorithm as an iterative expression

To develop a processor array, first we must be able to derive have to describe the string matching algorithm using recursions that convert the algorithm into of an RIA. We can write the basic string search algorithm as in Figure 3.1. This algorithm can also be expressed in the form of an iteration using two indices i and j.

$$y_i = \bigwedge_{j=0}^{m-1} \text{Match}(t_{i+j}, p_j), \qquad 0 \leq i \leq n-m \tag{3.3}$$

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

3.2.2 Obtaining the algorithm dependence graph (DG)

The string matching algorithm of Equation 3.3 is defined on a two-dimensional (2-D) domain since there are two indices (i, j). Therefore a data dependence graph can be

procedure *String_Search*(T, P)

$L1:$ **for** $i = 0$ **to** $n - m$

$L2:$ $\quad j \leftarrow 0$

$L3:$ $\quad$ **while** $(j < m \wedge t_{i+j} = p_j)$

$L4:$ $\quad\quad j \leftarrow j + 1$

$L5:$ $\quad$ **end while**

$L6:$ $\quad$ **if** $(j = m)$

$L7:$ $\quad\quad match\_flag \leftarrow TRUE$

$L8:$ $\quad\quad match\_location = i$

$L9:$ $\quad\quad$ **exit**

$L10:$ $\quad$ **endif**

$L11:$ **endfor**

end *String_Search*

Figure 3.1: The basic string search algorithm.

easily drawn as shown in Figure 3.2. The *computation domain* is the convex hull in the 2-D space where the algorithm operations are defined as indicated by the grayed circles in the 2-D plane [64]. 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

$$i = 3 \tag{3.4}$$

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

$$i + j = 3 \tag{3.5}$$

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

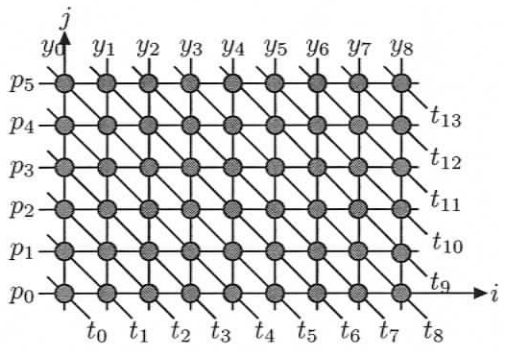


Figure 3.2: Dependence graph for $m = 4$ and $n = 10$

3.2.3 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 [64]

$$t(\mathbf{p}) = \mathbf{s}^t\mathbf{p} - s \tag{3.6}$$

where the function $t(\mathbf{p})$ associates a time value t to a point $\mathbf{p}$ in the DG. The column vector $\mathbf{s} = [s_1 \quad s_2]^t$ 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 Figure 3.2 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 Equation 3.6 to these

two points, we get

$$t(\mathbf{p}_1) = js_1 - s \tag{3.7}$$

$$t(\mathbf{p}_2) = (j+1)s_1 - s \tag{3.8}$$

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

$$\mathbf{s} = [1 \quad s_2]^t \tag{3.9}$$

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 [64]

$$\mathbf{s}^t\mathbf{e} \neq 0 \tag{3.10}$$

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

$$\mathbf{s}_1 = [1 \quad 1]^t \tag{3.11}$$

$$\mathbf{s}_2 = [1 \quad -1]^t \tag{3.12}$$

On the other hand, to broadcast a variable whose null-vector is $\mathbf{e}$, we must have [64]

$$\mathbf{s}^t\mathbf{e} = 0 \tag{3.13}$$

If we want to broadcast Y, then from Equation 3.13 and Equation 3.9, we must have

$$\mathbf{s}_3 = [1 \quad 0]^t \tag{3.14}$$

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 subsection.

3.2.4 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. Reference [64] explained how to perform the projection operation using a *projection matrix* **P**. To obtain the projection matrix we require to define a desired *projection direction* **d**. The vector **d** belongs to the null space of **P**. Since we are dealing with a two-dimensional DG, matrix **P** is a row vector and **d** is a column vector.

A valid projection direction **d** must satisfy the inequality [64]

$$\mathbf{s}^t\mathbf{d} \neq 0 \tag{3.15}$$

In the following three sections, we will discuss design space explorations for the three values of **s** obtained in Eqs. (3.11)-(3.14)

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

The feeding point of t_0 is easily determined from Figure 3.2 to be $\mathbf{p} = [0 \quad 0]^t$. Time value of this point is $t(\mathbf{p}) = 0$. Using Equation 3.6, we get $s = 0$.

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

$$\mathbf{e}_P = [1 \quad 0]^t \tag{3.16}$$

$$\mathbf{e}_T = [-1 \quad 1]^t \tag{3.17}$$

The product of **s** and these two null-vectors gives

$$[1 \quad 1]\,\mathbf{e}_P = 1 \tag{3.18}$$

$$[1 \quad 1]\,\mathbf{e}_T \;=\; 0 \tag{3.19}$$

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

There are three simple projection vectors such that all of them satisfy inequality (3.15) for the scheduling function in Equation 3.11. The three projection vectors will produce three designs

$$\text{Design 1.a:} \quad \mathbf{d}_a \;=\; [1 \quad 0]^t \tag{3.20}$$

$$\text{Design 1.b:} \quad \mathbf{d}_b \;=\; [0 \quad 1]^t \tag{3.21}$$

$$\text{Design 1.c:} \quad \mathbf{d}_c \;=\; [1 \quad 1]^t \tag{3.22}$$

The corresponding projection matrices could be given by

$$\mathbf{P}_a \;=\; [0 \quad 1]^t \tag{3.23}$$

$$\mathbf{P}_b \;=\; [1 \quad 0]^t \tag{3.24}$$

$$\mathbf{P}_c \;=\; [1 \quad -1]^t \tag{3.25}$$

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

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

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

$$\mathbf{p}' \;=\; \mathbf{P}_a^t \mathbf{p} = j \tag{3.26}$$

The processor array corresponding to Design 1.a is shown in Figure 3.3. 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

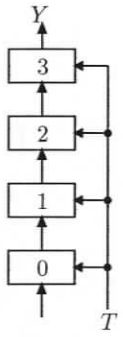


Figure 3.3: Processor array for Design 1.a when $\mathbf{s} = [1 \quad 1]^t$, $\mathbf{d}_a = [1 \quad 0]^t$, and $m = 4$.

array consists of m PEs and each PE is active for n time steps.

The PE details are shown in Figure 3.4 , where 'D' denotes a 1-bit register to store the output.

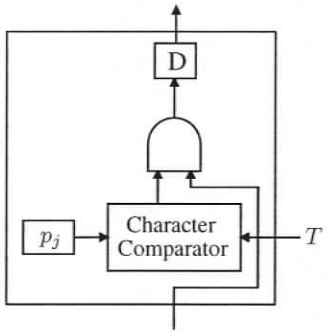


Figure 3.4: PE detail for Design 1.a in Figure 3.3.

3.3.2 Design 1.b: Using $\mathbf{s} = [1 \quad 1]^t$ 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

$$\mathbf{p}' = \mathbf{P}_b^t \mathbf{p} = i \tag{3.27}$$

The resulting processor array is shown in Figure 3.5.

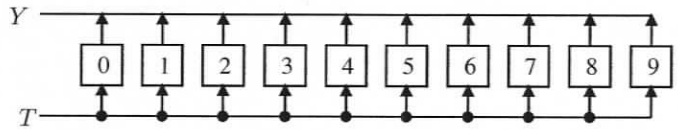


Figure 3.5: Processor array for Design 1.b when $\mathbf{s} = [1 \quad 1]^t$, $\mathbf{d}_b = [0 \quad 1]^t$, and $m = 4$.

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 as shown. Each PE is active for m time steps only. Thus the PEs are not well utilized as in the design of Section 3.3.1. However, we note from the DG of Figure 3.2 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

$$i' = i \bmod m \tag{3.28}$$

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 Figure 3.5 is shown in Figure 3.6. 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

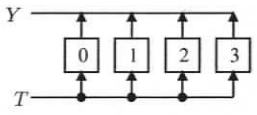


Figure 3.6: Processor array for Design 1.b after applying the modulo operation in Equation 3.28 for the case when $m = 4$.

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 Figure 3.7. A tristate buffer at the output of that PE ensures that it is the only output fed to the output bus as shown in Figure 3.7. The D register stores the output.

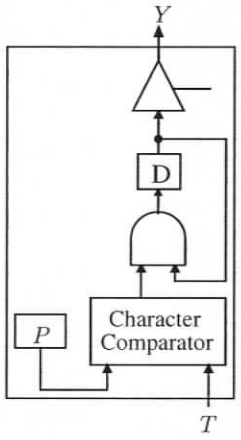


Figure 3.7: Processing element for Design 1.b in Figure 3.6.

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

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

$$\mathbf{p}' = \mathbf{P}_c^t \mathbf{p} = i - j \tag{3.29}$$

The resulting processor array is shown in Figure 3.8 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.

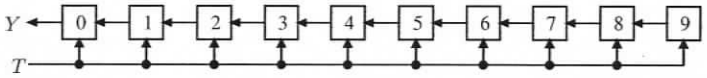


Figure 3.8: Processor array for Design 1.c when $\mathbf{s} = [1 \quad 1]^t$, $\mathbf{d}_c = [1 \quad 1]^t$, $n = 10$, and $m = 4$.

The processor array consists of n PEs where only m of the processors are active at a given time step as shown in Figure 3.9. At time step i, input text t_i is broadcast to all PE in the array.

We notice from Figure 3.9 that at any time step only m out of the n processors is active. To improve PE utilization, we need to reduce the number of processors. An obvious processor allocation scheme could be derived from Figure 3.9. In that scheme, operations involving the pattern word p_i are allocated to processor i. In that case the processor array in Figure 3.3 will result.

3.3.4 Comparing Design 1.a with Design 1.b

In the above subsections we derived two design options. Design 1.a in SubsectionSection 3.3.1 performs better than Design 1.b in Section 3.3.2 for the following reasons:

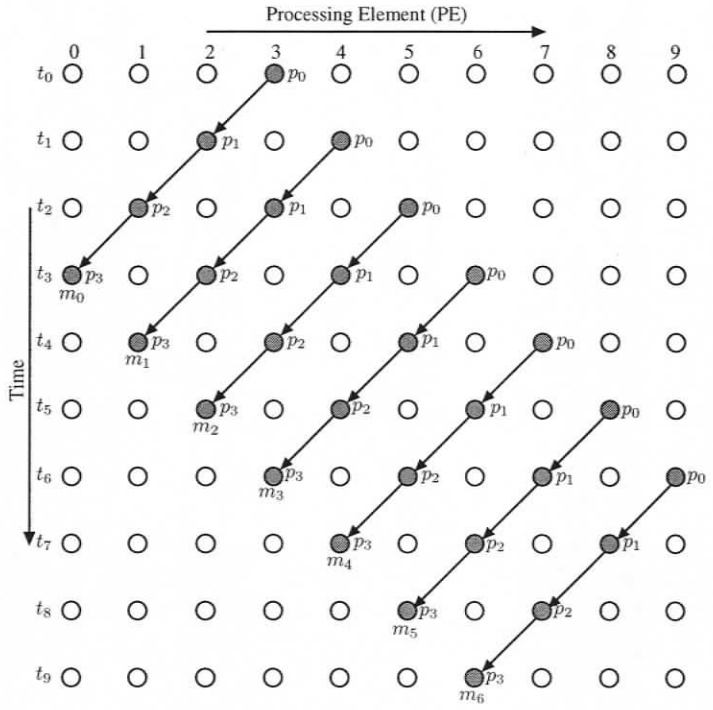


Figure 3.9: Processor activity at the different time steps for the design in Figure 3.8.

- PE_j of Design 1.a (shown in Figure 3.4) 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 Figure 3.7) stores the entire pattern P using on-chip memory module with its associated memory access delay [137].

- The clock period of Design 1.a (Figure 3.3) is given by,

$$\tau_{\text{clk}}(1\text{a}) = \max\left[\tau_p + \tau_d,\ \tau_b\right] \tag{3.30}$$

where, τ_p is the processing delay, τ_d is output driver delay, and τ_b of Equation 3.30 is the bus delay. τ_d is given by

$$\tau_d = \tau_0 \frac{C_l}{C_g} \tag{3.31}$$

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 [138, 139].

The bus delay τ_b is given by [140]

$$\tau_b = RC \times \frac{m(m+1)}{2} \approx 0.5\ RC\ m^2 \tag{3.32}$$

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.

The clock period of Design 1.b (Figure 3.5) is given by,

$$\tau_{\text{clk}}(1\text{b}) = \tau_p + \tau_b + \tau_m \tag{3.33}$$

where, τ_m is the memory access delay.

Typically τ_d smaller than τ_b which varies quadratically with the size of the processor array. Thus Design 1.a has 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

$$\varrho(1\text{a}) = m\varrho_{PE} + \varrho_b \tag{3.34}$$

where, ϱ_{PE} is the power consumption by each PE and ϱ_b is the power consumption of driving the bus.

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

$$\varrho(1\text{b}) = m\varrho_{PE} + 2\varrho_b \tag{3.35}$$

From Eqs. (3.34) and (3.35), Design 1.b consumes more power than Design 1.a.

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

3.4 Design 2: Design Space Exploration when s = $[1 \quad -1]^t$

Applying the scheduling function in Equation 3.12 to $\mathbf{e}_P$ and $\mathbf{e}_T$, we get

$$[1 \quad -1]\,\mathbf{e}_P = 1 \tag{3.36}$$

$$[1 \quad -1]\,\mathbf{e}_T = 2 \tag{3.37}$$

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 Figure 3.2. 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 $\mathbf{p} = [-m \quad m]^t$. The time value associated with this point is given by

$$t(\mathbf{p}) = -2m - s = 0 \tag{3.38}$$

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 3.3.1.

There are three simple projection vectors such that all of them satisfy inequality (3.15) for the scheduling function in Equation 3.12. The three projection vectors are

$$\text{Design 2.a:} \quad \mathbf{d}_a = [1 \quad 0]^t \tag{3.39}$$

$$\text{Design 2.b:} \quad \mathbf{d}_b = [0 \quad 1]^t \tag{3.40}$$

$$\text{Design 2.c:} \quad \mathbf{d}_b = [1 \quad -1]^t \tag{3.41}$$

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

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

Using the same treatment in Subsection 3.3.1, the resulting processor array is shown in Figure 3.10 for the case when $n = 10$ and $m = 4$.

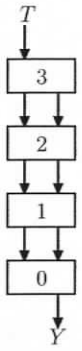


Figure 3.10: Processor array for Design 2.a when $\mathbf{s} = [1 \quad -1]^t$, $\mathbf{d}_a = [1 \quad 0]^t$, and $m = 4$.

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

The resulting processor array is similar to Design 1.b in Section 3.3.2. Thus it has the same problems as described in Section 3.3.4.

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

The resulting processor array is similar to Design 1.c in Section 3.3.3. Thus it has the same problems as described in Section 3.3.4.

3.4.4 Comparing Designs 2.a, 2.b and 2.c

In the above subsections we derived three design options. All three designs have a latency of $2m$ clock periods before the first result appears. Design 2.a in Subsection

3.4.1 requires the least area since it does not require on-chip memory to store the pattern P. In summary, Design 2.a is the best among the three designs from the point of view of area.

3.4.5 Comparing Designs 1.a and 2.a

The clock period of Design 2.a is given by

$$\tau_{\text{clk}}(2\text{a}) = \tau_p + \tau_d \tag{3.42}$$

Typically $\tau_p + \tau_d < \tau_b$, Design 2.a is faster than Design 1.a.

Power consumption in Design 2.a is given by

$$\varrho(2a) = m\varrho_{PE} \tag{3.43}$$

Comparing this equation with Equation 3.34, 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.

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

The feeding point of t_0 is easily determined from Figure 3.2 to be $\mathbf{p} = [0 \quad 0]^t$. Time value of this point is $t(\mathbf{p}) = 0$. Using Equation 3.6, we get $s = 0$.

Applying the scheduling function in Equation 3.11 to $\mathbf{e}_P$ and $\mathbf{e}_T$, we get

$$[1 \quad 0]\,\mathbf{e}_P = 0 \tag{3.44}$$

$$[1 \quad 0]\,\mathbf{e}_T = -1 \tag{3.45}$$

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 inequality (3.15) for the scheduling functions in Equation 3.11. These projection vectors are

$$\text{Design 3.a:} \quad \mathbf{d}_a = [1 \quad 0]^t \tag{3.46}$$

$$\text{Design 3.b:} \quad \mathbf{d}_b = [1 \quad 1]^t \tag{3.47}$$

$$\text{Design 3.c:} \quad \mathbf{d}_c = [1 \quad -1]^t \tag{3.48}$$

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

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

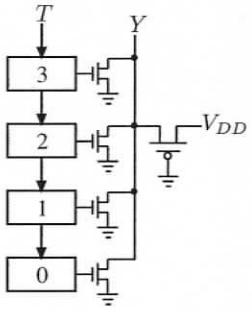


Figure 3.11: Processor array for Design 3.a when $\mathbf{s} = [1 \quad 0]^t$, $\mathbf{d}_a = [1 \quad 0]^t$, and $m = 4$.

The processor array corresponding to Design 3.a is drawn in Figure 3.11. 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 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 form 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 3.3.4. 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. The design has a latency of only one time step. This is the least latency exhibited by design reported here or previously proposed so far.

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

These two projection vectors produce the same processor array. Each PE stores the entire patter P in on-chip memory. The resulting array is similar to Designs 1.c and 2.c but all PEs operate on one output value at the same time. The design has a latency of only one time step. However, the time step value is limited by the maximum of the access time for the on-chip memory or the propagation delay of the system-wide bus, similar to Designs 1.b, 2.b, and 2.c.

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

Design 3.a has the smallest area since it does not require on-chip memory. In summary, Design 3.a is the best among the three designs from the point of view of area.

3.5.4 Comparing Design 3.a with Design 1.a or 2.a

Design 3.a is 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 bus propagation delay. That bus is the output line connecting the driver transistors of the NOR gate. Thus the clock period of Design 3 is given by

$$\tau_{\text{clk}}(3\text{a}) = \tau_p + \tau_{NOR} \approx \tau_p + \tau_b \tag{3.49}$$

which is bigger than the clock periods given in Equation 3.30 or Equation 3.42. Thus, Design 3.a is slower than Design 1.a or Design 2.a. So, Design 2.a is the best design among the nine designs proposed in this chapter.

3.6 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 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 Design 3 is determined by bus propagation delay that increases quadratically with the number of PEs.

3.6.1 Best Case

In the best case y_0 will indicate a match. This output is obtained after m time steps in Designs 1.a and 3.a and after $2m$ time steps in Design 2.a.

3.6.2 Worst Case

In the worst case, all y_i outputs with $0 \leq i < m - n$ 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 for Designs 1 and 3 and after $n + m$ time steps for Design 2.

3.6.3 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

$$p_m = \frac{1}{|\Sigma|} = \frac{1}{2^w} \tag{3.50}$$

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 $0 \leq j < i$ produced negative results. We can express α_i as

$$\alpha_i = p_m^m \left(1 - a_m^m\right)^{i-1} \tag{3.51}$$

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

$$T_{av} = \sum_{i=0}^{n-m} (m+i)\ \alpha_i \tag{3.52}$$

After a rather laborious algebraic manipulation (see [65]), we obtain

$$T_{av} = n - m \tag{3.53}$$

3.7 Simulations

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 3.6.

We perform the simulations based on the following assumptions:

- Number of simulations = 100,000.
- $w = 32$ for typical 32-bit machine.

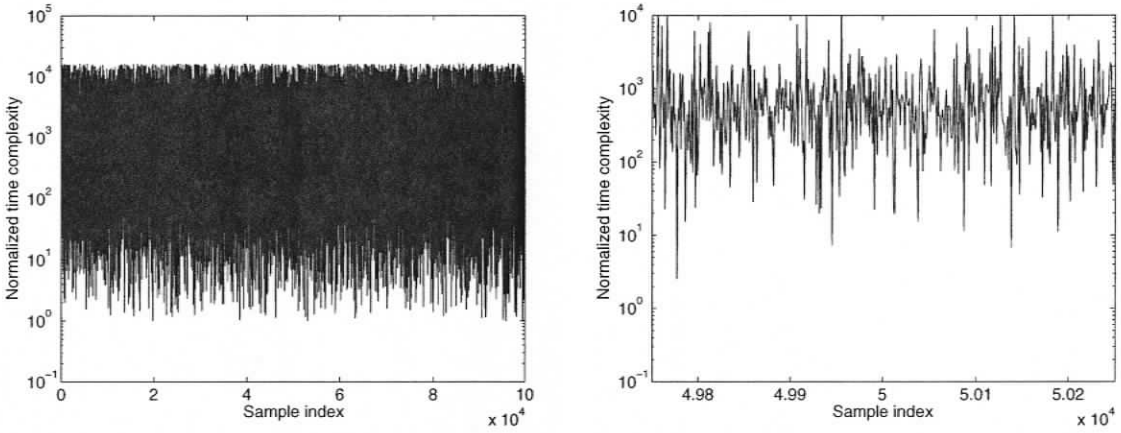
- Maximum value of n is 16,384 (which corresponds to the maximum network packet size).
- Maximum value of m is 25.
- P, T, m, and n are randomly generated. We use uniform distribution so that each value is equally likely.

3.7.1 Best Case

The best case time complexity derived in Section 3.6 is m. Since m has been varied randomly in each simulation, we normalize each result by the corresponding m. Figure 3.12(a) shows the graph of the normalized value vs. sample index. Figure 3.12(b) shows the normalized results for 500 simulations taken from the middle of Figure 3.12(a). This allows us to see the fine scale variations. In Figure 3.12, the minimum normalized value is 1. Thus the best case time complexity is m as analytically derived in Section 3.6.

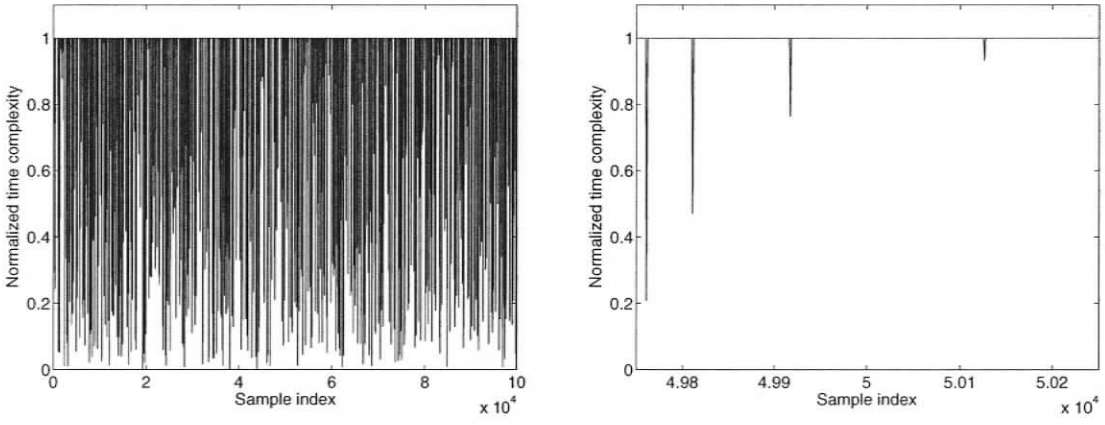
3.7.2 Worst Case

The worst case time complexity derived in Section 3.6 is n. Since n has been varied randomly in each simulation, we normalize each result by the corresponding n. Figure 3.13(a) shows the graph of the normalized value vs. sample index. Figure 3.13(b) shows the normalized results for 500 simulations taken from the middle of Figure 3.13(a). This allows us to see the fine scale variations. In Figure 3.13, the maximum normalized value is 1. Thus the worst case time complexity is n as derived in Section 3.6.



(a) (b)

Figure 3.12: Experimental results are plotted in (a) for all simulations and in (b) for 500 simulations. Results are normalized by m.



(a) (b)

Figure 3.13: Experimental results are plotted in (a) for all simulations and in (b) for 500 simulations. Results are normalized by n.

3.7.3 Average Case

The average case time complexity derived in Section 3.6 is $n - m$. Like Section 3.7.1 and Section 3.7.2, we normalize each result by the corresponding $n-m$. Figure 3.14(a) shows the graph of the normalized value vs. 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 3.6. Figure 3.14(b) is the histogram of the results of Figure 3.14(a). This figure shows that almost all the normalized results lie in the range between 0 and 3. So we redraw the histogram in Figure 3.14(c) in the range between 0 and 3. In Figure 3.14, 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 3.6.

3.8 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.

We employed a systematic technique to obtain our processor arrays by first converting the string search algorithm to an RIA. Having obtained the RIA, were were able to develop a 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 Section 3.4.1 is identical to the one obtained by Foster and Kung [131] using ad-hoc techniques. Design 2.a is also similar to that proposed by Park and George [133]. Similar processor array has also been derived by Sastry,

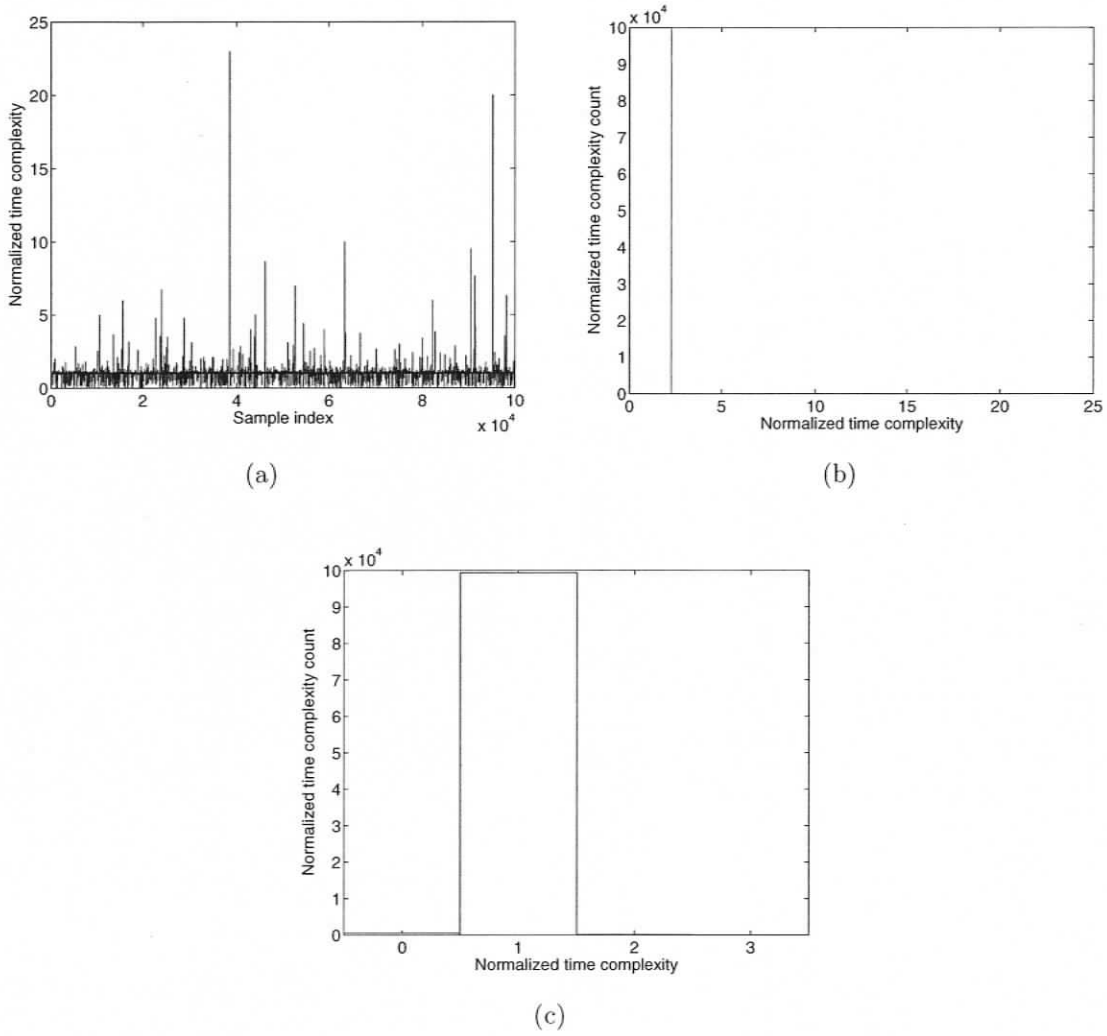


Figure 3.14: Experimental results are plotted in (a). Figure 3.14(b) is the histogram of the results of Figure 3.14(a). Figure 3.14(c) is the same histogram in the range from 0 to 3. Results are normalized by $n - m$.

Ranganathan, and Remedios in [136].

The processor array of Mukherjee [132] determines the similarity between two strings instead of finding exact matches. Our systematic technique could be easily

adapted for this situation by properly modifying Equation Equation 3.3. However, the design proposed in [132] 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 $\mathcal{O}(n-m)$ (Section 3.6). Our design approach could also be adapted to implement the approximate text searching considered by Michailidis and Margaritis [134, 135]

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.

3.9 Conclusion

This chapter 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 are 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 $n-m$ (Equation 3.53). We also compare our processor arrays with previously derived processor arrays for the string matching problem.

Chapter 4

Embedding Techniques for Parallel Solution

In this chapter, we propose a novel technique that embeds a source processor array onto a target processor array having smaller number of PEs such that the dimensions of the two arrays are the same. For example, a two-dimensional array is embedded into a two-dimensional array of smaller size. We have applied our embedding technique on one of the best architectures for deep packet classification that we previously obtained in Section 3.

This chapter is organized as follows: Section 4.1 discusses the literature related to some processor array implementations that take FPGA resource constraints (area, I/O, etc.) into account. Section 4.2 discusses our proposed embedding technique. Section 4.3 discusses the embedded or target processor array for DPC. Section 4.4 analyzes the performance of the target architecture using Markov model analysis. Section 4.5 discusses the synthesized results of the target architecture on the Stratix II Altera FPGA and shows the effects of number of PEs on FPGA parameters. Section 4.6 discusses the simulation results of the C-model based on the target architec-

ture. Finally we conclude our chapter in Section 4.7.

4.1 Existing Works

In [65], we summarized some processor array implementations by other authors for the string matching problem. In this section, we summarize only those processor array implementations that take FPGA resource constraints (area, I/O, etc.) into account.

In [141], Teich et al. used an objective function to optimize the execution time using the available resources. Before this optimization process, they used *tiling* to embed variables in higher dimensional index spaces into the resource-constrained computational domain. Tiling increases the granularity of computations and the locality of data references. Also the performance of the hardware is parameter-dependent. Using their technique, once an optimal hardware is derived for certain input parameters, the hardware cannot dynamically adapt with changes in input parameters.

In [142], Ganapathy used an approach to that in [141]. But unlike *tiling*, Ganapathy partitioned the computational space according to the design constraints to reduce the communication between each partition. References [143,144] also proposed similar partitioning techniques.

In [145], Iriogoin et al. proposed a partition method, named 'Supernode Partition', to group computations (DO loops) to form blocks (supernodes) such that these nodes can be executed in parallel on supercomputers or any multiprocessor system. Due to data locality, this technique improves the performance of the system to a great extent. However, it is difficult to determine the size of supernodes to be mapped onto a processor array. The performance of the resultant hardware is also parameter-dependent.

In [146, 147], Shang et al. used Hermite Normal Form to map n-dimensional

algorithms into $(k-1)$-dimensional arrays without computational conflicts where $k < n$. The computational conflict occurs when two or more computations/nodes in same isotemporal planes are mapped into the same processor. Their approach does not guarantee to perform optimally in all cases.

In [59, 148, 149], two methods were used to partition the computational domain: (i) locally serial globally parallel (LSGP) method and (ii) locally parallel globally serial (LPGS) method. In the LSGP method, one partition (block) is mapped to one PE. Each PE sequentially executes the nodes of the corresponding block. In the LPGS scheme, one block is mapped to one array. All nodes within a block are processed concurrently. One block after another block of node data are loaded into the array and processed in a sequential manner. In [150], Kim et al. used LPGS to partition the computational domain for GF(2^m) multiplier. But in [151], Chen et al. proved that LSGP and LPGS methods do not guarantee efficient data communication between partitions. Instead, they formed a supernode like [145] with n partitioning vectors such that the data flow in the supernode domain is very local and orientated along the positive direction in every dimension.

In [152], Koziris et al. described Chain Grouping, a similar method like [145], to partition the loop iteration space into groups with little intercommunication requirements between partitions. Inside every group, the optimal hyperplane scheduling is applied. Like [145], this partitioning method does not guarantee optimal performance in all cases.

In [153, 154], Yamada et al. proposed a string-search engine comprised of finite-state automaton (FSA) logic and content addressable memory (CAM). A special CAM called PCAM (pair-bit CAM) is used to accommodate variable-length patterns. In this PCAM, character comparisons are performed concurrently. Based on these comparison results, FSA searches for both exact and approximate matching. Although this hardware can handle variable-length patterns, it cannot operate on

arbitrary-length patterns (maximum length limitation).

The common problem of the above mentioned techniques is that the execution time of the processor arrays is much higher than that of arrays produced without resource-constraints. On the other word, the execution of the target array is much higher than that of the source array. In this chapter, we develop a novel embedding technique that has the following favorable characteristics:

- the proposed technique is simple to apply,
- interconnection patterns among PEs of the source and target arrays are same,
- execution times of the source and target processor arrays are same,
- memory accesses of the source and target processor arrays are same, and
- I/Os of the target array are simple (no multiplexed I/O).

Our proposed technique is illustrated using the pattern matching algorithm for DPC as a running example. Reference [65] explored nine possible processor array architectures. The resulting processor arrays are characterized by (i) the number of PEs, (ii) the functionality of each PE, (iii) the interconnections among PEs, and (iv) the I/O location and timing. The number of PEs is the most important characteristic, since it determines the hardware resources of the system and determines all other characteristics mentioned above. But in the derived architecture, the number of PE is determined by the input parameters. To make the number of PEs of the array to be independent of the input parameters, we will propose a novel embedding technique in the following section.

4.2 Proposed Embedding Technique

The idea behind the proposed embedding technique is to use non-linear projection operation instead of the linear projection operation given in Equation 3.26. According to the new projection operation, each node $\mathbf{p}$ of the k-dimensional DG is projected onto another point $\mathbf{r}$ in a k_1-dimensional space. Since each point corresponds to a PE, the dimensionality of the target array is the same as the source array.

$$\mathbf{r} = \min(\mathbf{Pp}, \mathbf{a}) \tag{4.1}$$

where $\mathbf{a}$ is a vector of available PEs in each dimension.

$$\mathbf{a} = [a_0, a_1, \cdots, a_{K_1-1}]^t \tag{4.2}$$

$\mathbf{r}$ can be expressed as,

$$\mathbf{r} = [r_0, r_1, \cdots, r_{K_1-1}]^t \tag{4.3}$$

Based on Equation 3.26, we can re-write Equation 4.1 as

$$\mathbf{r} = \min(\mathbf{q}, \mathbf{a}) \tag{4.4}$$

Equation 4.4 can be expanded as:

$$r_0 = \min(q_0, a_0) \tag{4.5}$$

$$r_1 = \min(q_1, a_1) \tag{4.6}$$

$$\vdots$$

$$r_{k_1-1} = \min(q_{k_1-1}, a_{k_1-1}) \tag{4.7}$$

Figure 4.1 illustrates the application of Equation 4.4 for the case of two-dimensional source array ($k_1 = 2$). Figure 4.1(a) shows 6×6 source processor array which consists of total 36 PEs. Assume that the available resources can accommodate only 16 PEs. In that case, our target processing array will have to be of size 4×4, i.e., $\mathbf{a} = [3 \quad 3]^t$.

To meet this constraint, we apply Equation 4.5 to Figure 4.1(a) and get the processor array as drawn in Figure 4.1(b). For example, points in group G_0 are mapped to a single point, G_0 in Figure 4.1(b). To continue the procedure, we apply Equation 4.6 to Figure 4.1(b) and get the processor array as drawn in Figure 4.1(c). For example, points in group G_6 are mapped to a single point, G_6 in Figure 4.1(c).

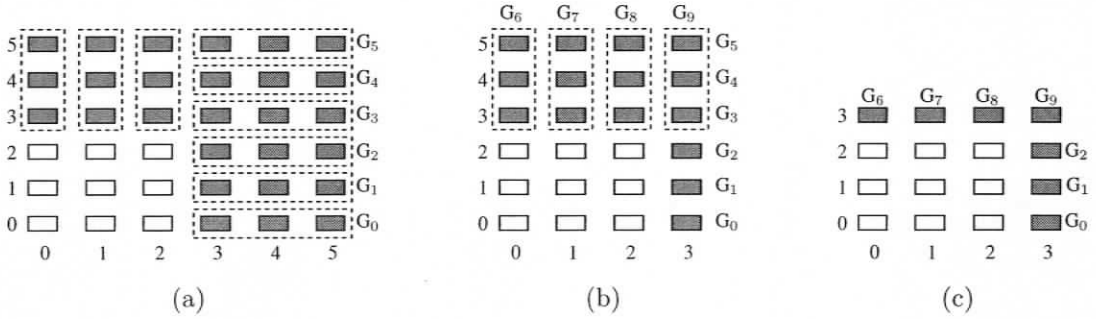


Figure 4.1: Illustration of Equation 4.4 for the case when $k_1 = 2$ and $\mathbf{a} = [3 \quad 3]^t$: (a) shows the point of $\mathbf{q}$ and (c) shows the point of $\mathbf{r}$.

The proposed embedding technique impacts the four factors mentioned in Section 1.4.3 in the following way. Since the highest indexed PE in target array corresponds to several PEs in the source array, it linearly increases the execution time. The highest indexed PE has to access the memory more than that of the other PEs. Since the highest indexed PE has to do more operations than other PEs, the intermediate results may have to be stored for proper operation. If the arrival of inputs depend on time, some inputs may have to be stored by the highest indexed PE. However, if the scheduling and projection operations can ensure that each dependent data is processed in separate PE, then the workload of the highest indexed PEs can be close to one and the memory access by the highest indexed PEs can be minimum. To take advantage of this, the algorithm, to be implemented in the processor array, must have the feature that the probability to continue the algorithm decreases as the algorithm

is being executed. These will be further discussed in the following section.

Interconnection among the PEs will not be affected by the proposed embedding technique. However, one or two more I/Os may be required for the highest indexed PE. But I/O multiplexing is not required which may reduce the execution time.

4.3 Embedded Processor Array Designs

We explored nine possible processor array architectures for DPC in [65] using the projection operation given Equation 3.26. As a running example for our proposed embedding technique, we pick up one architecture shown in Figure 4.2(a), where variables P and Y are pipelined, T is broadcast, and $k_1 = 1$ as the source architecture (details can be found in [65]). Since $k_1 = 1$, $\mathbf{q}$, $\mathbf{r}$, and $\mathbf{a}$ become q, r, and a (i.e., scalar) respectively for the architecture in Figure 4.2(a). This architecture has a non-favorable characteristic: the number of PEs depends on m, i.e.,

$$q_{\max} = m - 1 \tag{4.8}$$

Instead, if we use the Equation 4.4 for the projection operation, we get the architecture as shown in Figure 4.2(b), where

$$r_{\max} = a \tag{4.9}$$

a is a user-defined parameter determined based on resource constraints, whereas m is the number of characters in P and varies from application to application. Thus the embedded or target architecture eliminates the problem posed by the source architecture.

In Figure 4.2, both source and target arrays look alike except the number of PEs. however their activities differ from each other. We draw the activities of both source and target processor arrays in Figure 4.3. Figure 4.3(a) shows the activities of the

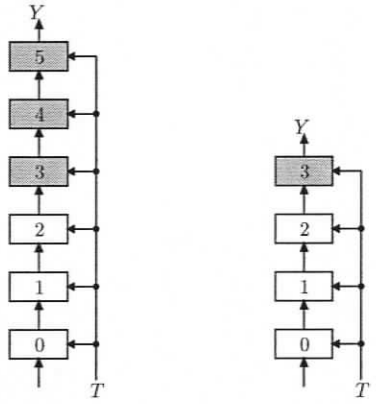


Figure 4.2: Processor array architectures: (a) source array for $m = 6$, (b) target array for $a = 3$.

array in Figure 4.2(a). The arrows show the flows of the intermediate results between the PEs. A node calculates $A_{i,j}$ which is defined by the following recursive equation:

$$A_{i,j} = \begin{cases} C_{i,j} \wedge A_{i-1,j-1} & \text{when } j \geq 0 \\ \text{TRUE} & \text{otherwise} \end{cases} \tag{4.10}$$

Thus each PE is responsible for two operations:

1. compare between t_i and p_j, i.e. $C_{i,j}$ calculation, and

2. AND operation between $C_{i,j}$ and $A_{i-1,j-1}$ which is previously calculated.

Since the $A_{i,j}$ calculation requires $A_{i-1,j-1}$ which is calculated in the lower indexed PE, PE_a doing the operations $A_{I,J}$ (where $a \leq I < n$ and $a \leq J < m$) can preempt any more operation for $A_{I,J}$ if the result of $A_{I-1,J-1}$ is FALSE. This preemption lowers the workload of PE_a.

We draw the activities of the target processor array at different time indices in Figure 4.3(b), where PE_j ($0 \leq j < 3$ or in general $0 \leq j < a$) does its normal

operation like Figure 4.3(a), but PE_a does maximum 3 (or in general $m-a$) operations at a given time index. For example, PE_3, at time index 5, produces: (i) $A_{5,5}$, (ii) $A_{5,4}$, and (iii) $A_{5,3}$. Out of these 3 outputs, $A_{5,5}$ produces y_0. The other two (or in general $m-a-1$) results must be saved in a queue.

Each PE of the target processor array in Figure 4.2(b) looks like Figure 4.4. However, memory requirements and controller complexity are much higher for PE_a than those of other PEs. The PE details are described below:

- I/Os are required for communication between PEs, data input/output, and control input.
- PE_a requires a queue, Q_1 to store comparison results. From Figure 4.3(b), maximum $m-a-1$ comparison results have to be stored for future use. To speed up the matching operation, if there is any mismatch, then PE_a will not try to find any other match for that particular y_i.
- PE_a requires another queue, Q_2 to store characters of T. Since Q_1 stores $m-a-1$ comparison results, we do not need to store more than $m-a-1$ characters of T. So, the size of Q_2 is $m-a-1$ as well. The queued characters can be used by PE_a in later time indices, if they are required.
- PE_a requires a counter to count the number of times, the PE_a has to calculate Y.
- Each PE requires a simple ALU capable of basic numerical, logical, and boolean operations.

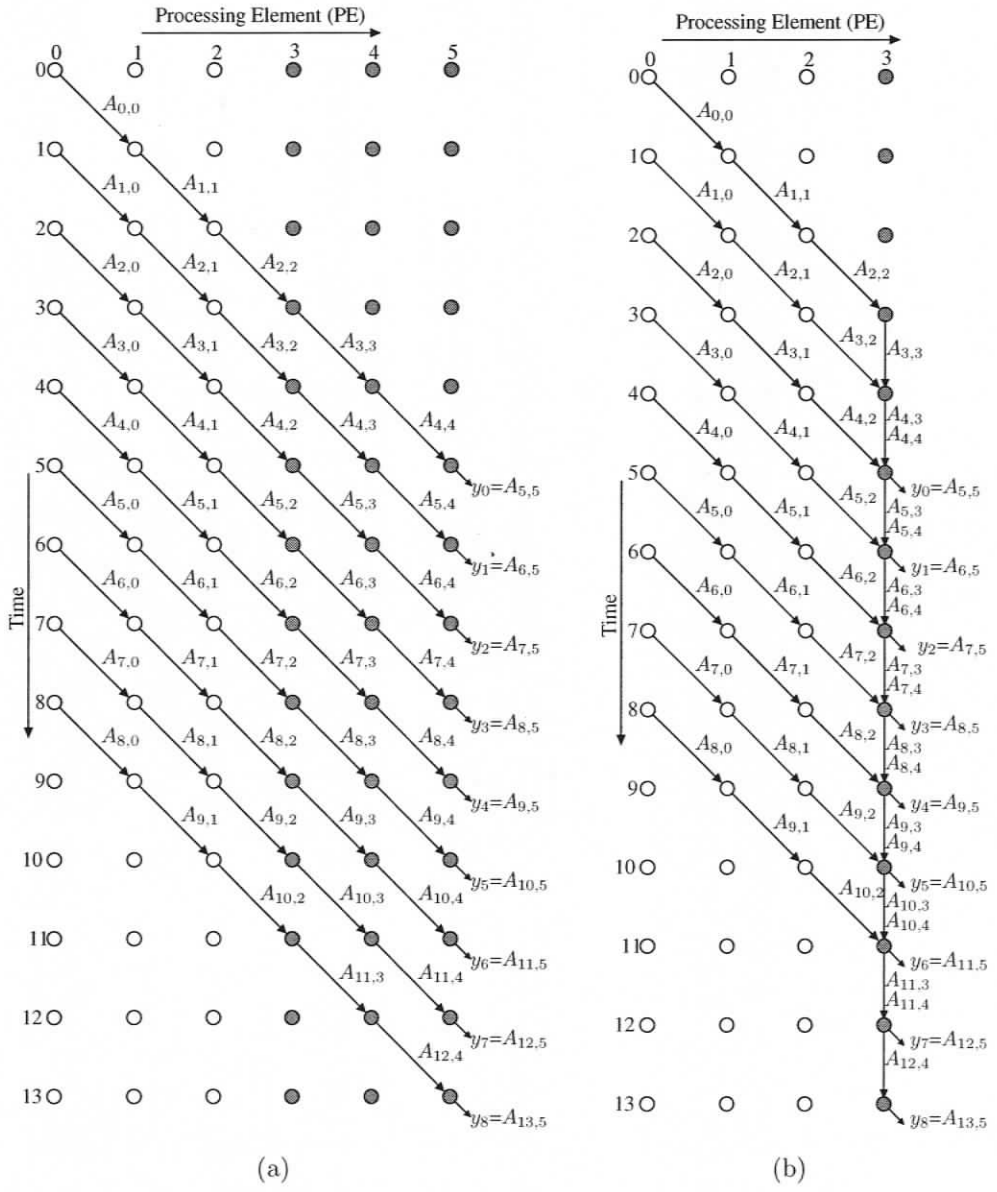


Figure 4.3: Activity of the processor array for Design 1: (a) source array for $m = 6$, (b) target array for $m = 6$ and $a = 3$.

4.4 Time Complexity Analysis

In this section, we determine the performance of the embedded architecture in Figure 4.2(b). We develop a Markov model for PE_a of the array, because the states

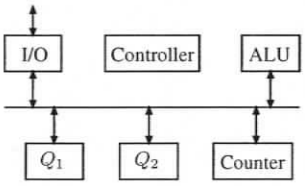


Figure 4.4: PE of the embedded processor array in Figure 4.2(b)

of PE_a exhibits memoryless property as required for the Markov model [155]. Figure 4.5 shows the Markov chain transition diagram. The Markov model has following assumptions:

1. Each state represents a task ($A_{i,j}$ calculation) executed by PE_a.
2. Each task requires one time step to be executed.
3. There are total $(m-a) \times n + 2$ states in the state vector ($(m-a) \times n$ tasks to be done by PE_a plus Match and Mismatch states).

The associated transition matrix for Figure 4.5 is given by Equation 4.11. i and j of the transition probability, $p_{i,j}$ correspond to the column and row indices of the Equation 4.11, respectively.

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ p_m^{a+1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & p_m & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1-p_m^{a+1} & 1-p_m & 1-p_m & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & p_m^{a+1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & p_m & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-p_m^{a+1} & 1-p_m & 1-p_m & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & p_m^{a+1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & p_m & 0 & 0 & 0 \\ 0 & 0 & p_m & 0 & 0 & p_m & 0 & 0 & p_m & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1-p_m^{a+1} & 1-p_m & 1-p_m & 0 & 1 \end{pmatrix} \quad (4.11)$$

where, p_m is the matching probability between two characters. We assume equally likely distribution for p_m i.e.,

$$p_m = 2^{-w} \quad (4.12)$$

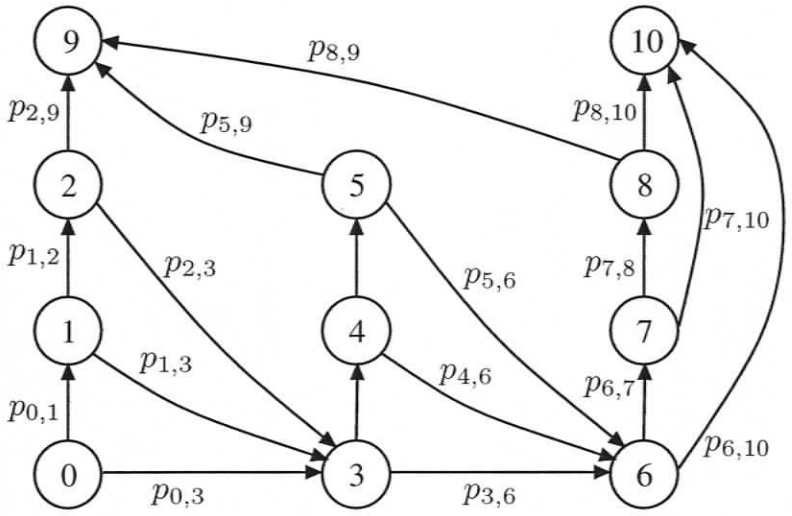


Figure 4.5: Transition diagram for PE_a of the target array in Figure 4.2(b) when $m = 6$, $n = 7$, and $a = 3$. States 9 and 10 represent Match and Mismatch states, respectively.

To calculate the average time (T_a) required to go to the Match and Mismatch states, we use the following equation [155],

$$T_a = \sum_{\eta} (a + 1 + \eta) \left(\mathbf{x}(\eta)_{\text{Match}} + \mathbf{x}(\eta)_{\text{Mismatch}} \right) \tag{4.13}$$

where,

$\mathbf{x}(\eta)_z$ = probability of being in state z at time step η.

To calculate the state vector $\mathbf{x}(\eta)$, we use the following equation [155],

$$\mathbf{x}(\eta) = \mathbf{A}^{\eta} \mathbf{x}(0) \tag{4.14}$$

where,

$\mathbf{x}(0)$ = state vector at time step $0 = [1\ 0 \cdots 0]^t$

Using Maple [156], we have got $T_a = 1$ assuming $nm \rightarrow \infty$. In [65], we derived the average execution time of the source processor array in Figure 4.2(a) to

be $\mathcal{O}(n-m)$. Since the average execution time of the PE_a of the target processor array is smaller than $n-m$, the average execution time of the target processor array in Figure 4.2(b) is also $\mathcal{O}(n-m)$. We will also prove this value using numerical simulation in Section 4.6.

4.5 Synthesis of the Embedded Processor Arrays

We synthesized our target processor array in Figure 4.2(b) for four reasons:

1. We want to verify the correctness of the operation of the target processor array on a FPGA.
2. We want to see the effects of the number of PEs in the target array on the different implementation parameters.
3. We want to verify the memory access and I/O complexity for the target array.
4. The synthesis results will be used in Section 4.6 to verify the average execution time of the target array using computer simulation.

To synthesize the target processor array, we used Altera Stratix II FPGA, because Altera is one of the leading FPGA producers and Stratix II is the fastest devices of Altera FPGA [157]. We used the following values in our synthesis:

- $n = 16{,}384w$ (which corresponds to the maximum IP network packet size),
- $m = 50w$, and
- $w = 4$ bytes.

We took 5 values of a to synthesize the target processor array. We ensure $m > a$ to see the effect of our embedding technique on the processor array. We implemented

the controller of the PE in Figure 4.4 as an FSM (Finite State Machine), and VHDL to describe the architecture in Figure 4.2(b). Before further discussion, we need to know the relationship between the number of PEs in the target array and a:

$$\text{number of PEs} = a + 1 \tag{4.15}$$

We discuss the synthesis results in the following.

Figure 4.6 shows the number of Logic Cell (LC) required for the target processor array versus a. LCs are used to implement ALU, controller, and the small memory and/or memory-based components like counter. Each PE consists of ALU and controller that linearly increases the LC usage as a increases, while the last PE (PE_a)

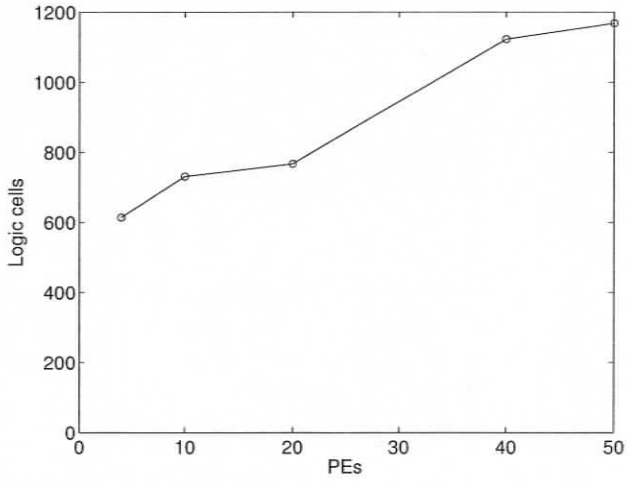


Figure 4.6: Effect of a (number of PEs) on the Logic Cell (LC) usage of the target processor array.

has a counter whose size logarithmically deceases as a increases. For these reasons, LC increases non-linearly as a increases in Figure 4.6.

Figure 4.7 shows the pin count for the target processor array versus a. This figure shows that increase of a by 1 causes the increase of the I/O pin requirement by

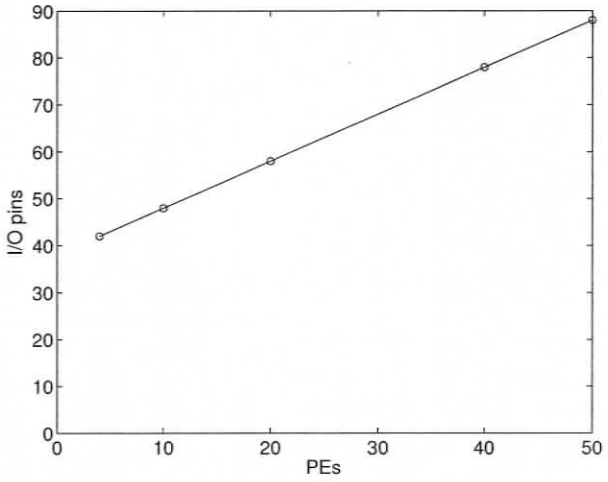


Figure 4.7: Effect of a (number of PEs) on the I/O pin requirement of the target processor array.

1, because each PE is connected with a control pin along with other I/Os. We can use decoded control pins to reduce the I/O pin requirement instead. Since decoded control pins do not have any impact on system speed, we used one control pin for each PE in our synthesis. But for area-critical systems, it is recommended to use decoded control pins. However, there is no multiplexed I/O in the target array that may degrade the performance of the array.

Figure 4.8 shows the memory usage for the target processor array versus a. Only PE_a requires two queues that are implemented on RAM. The sizes of these queues are $m-a-1$ that decease as a increases (for fixed m). For this reason, in Figure 4.8, the memory usage decreases by $2 \times (a_2 - a_1) \times w$ as a increases. For example from Figure 4.8, the memory requirements = 3,072 bits for $a = 3$. Then the memory requirements should be $3,072-2\times(9-3)\times 32 = 2,688$ bits for $a = 9$ as in Figure 4.8. Although memory requirements of the PE_a increase as $m - a - 1$ increases, memory access would be very low since the workload of the PE_a is very low as proved in

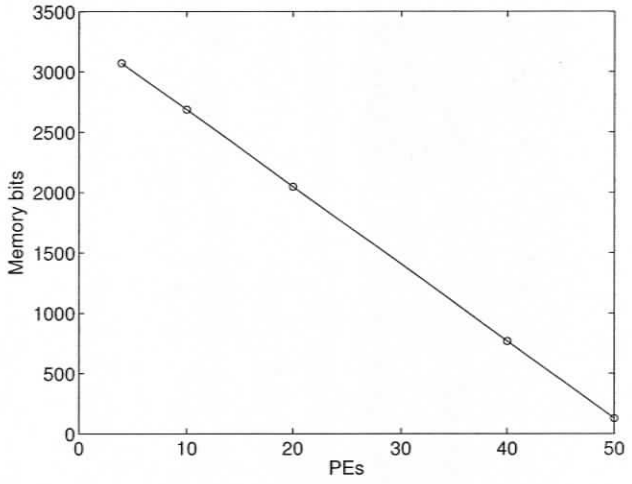


Figure 4.8: Effect of a (number of PEs) on the memory usage of the target processor array.

Section 4.3.

Figure 4.9 shows the longest path delay for the target processor array versus a. The time-critical path of the system in Figure 4.2(b) is the input bus as determined in [65]. As a increases, the length of the bus and the number of fan-in increase linearly. Since these two factors have quadratic impact on the critical path delay, the longest path delay quadratically increases as shown in Figure 4.9.

Figure 4.10 shows the clock period for the target processor array versus a. Since the time-critical path is not clock dependent, the clock period is not affected by this path. The clock period in our synthesis depends on the size of the queues. Queues are supported by some logic units that decrease logarithmically in their sizes as a increases. Since smaller-sized logic units require smaller clock period, the clock period decreases logarithmically as a increases (shown in Figure 4.10).

In practical situations, once the array is designed, the queue sizes would be fixed. In that case, the clock period and the memory requirements would be static. Since

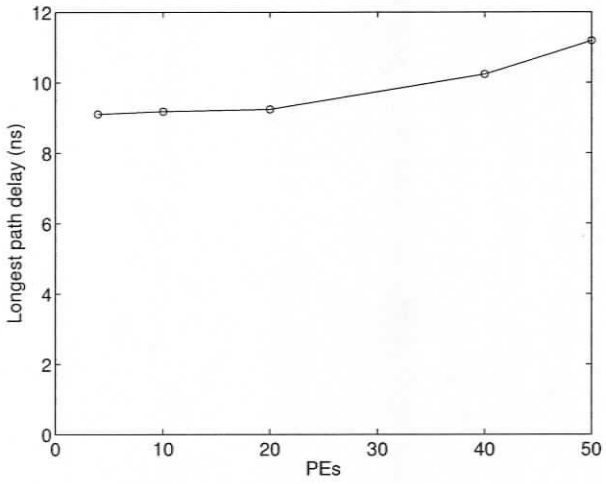


Figure 4.9: Effect of a (number of PEs) on the longest path delay of the target processor array.

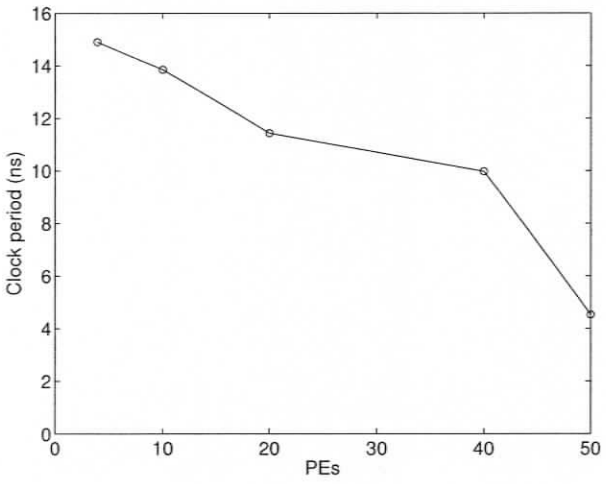


Figure 4.10: Effect of a (number of PEs) on the clock period of the target processor array.

our embedding technique plays with a, we are varying a (m remains fixed) to examine the effect of a on different FPGA parameters. However, in practical, a remains fixed,

while m varies.

4.6 Modeling and Simulations

We modeled the synthesized embedded processor array using C language. We simulated the C-model for five values of a: 3, 9, 19, 39, and 49. We did the simulation 10^5 times for each of these five values of a. Each time, we used the uniform distribution to randomly generate all characters of P and T and the lengths: m, n. We counted the number of time steps, C_t in each simulation and normalized the count by $n - m$ to get meaningful data. Using these values, we do the following calculations.

4.6.1 Time Complexity Calculation

Figure 4.11 is the result of experiments for $a = 3$. The plot of the experimental results in Figure 4.11(a) and Figure 4.11(b) show that normalized clock counts are close to 1 in most of the situations. The histogram in Figure 4.11(c) also proves this observation. So, we can say that the average execution time is $\mathcal{O}(n - m)$.

We draw the experimental results in Figs. 4.12–4.15 for $a = 9$, 19, 39, and 49, respectively. These figures also prove that the average execution time of the embedded hardware is $\mathcal{O}(n - m)$, as was derived in Section 4.4.

4.6.2 Throughput Calculation

To calculate the throughput of the target processor array, we multiply C_t with the corresponding clock period derived in Section 4.5 to get the execution time of the target processor array. We calculate the median of the execution time for five values of a. From these medians, the target processor array with 50 PEs shows the best performance. Since each PE works on 4 bytes of data at a time, 50 PEs mean that

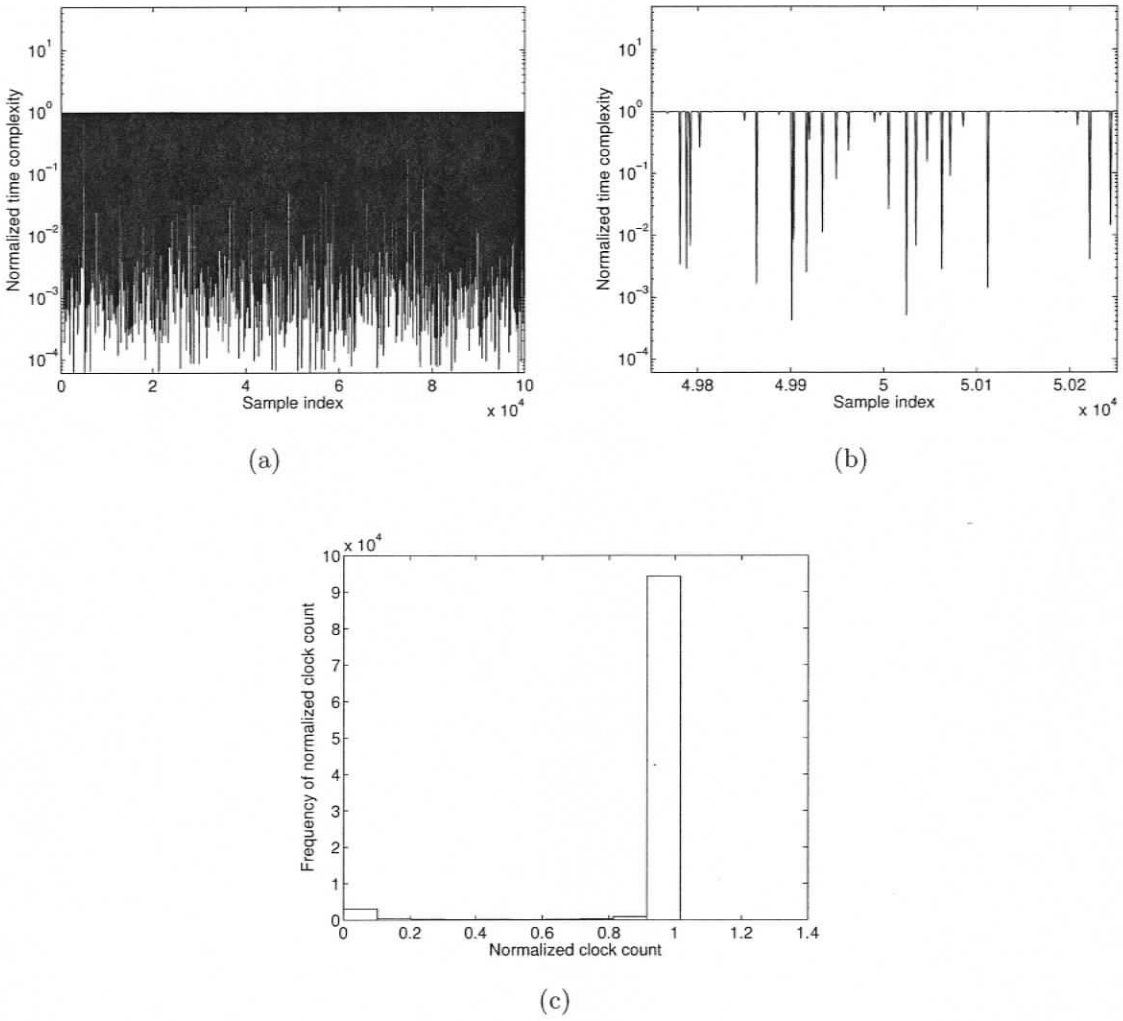


Figure 4.11: Experimental results for $\overline{m} = 4$: (a) for all data, (b) for 500 data, (c) the histogram.

the processor array can handle 200 (50×4) bytes which is equal to m. For this reason, we calculate the throughput of our embedded array when $a = 50$. So the throughput of our design is

$$\frac{4 \text{ bytes}}{4.522 \text{ ns}} = 26.362 \text{ Gbit/sec}$$

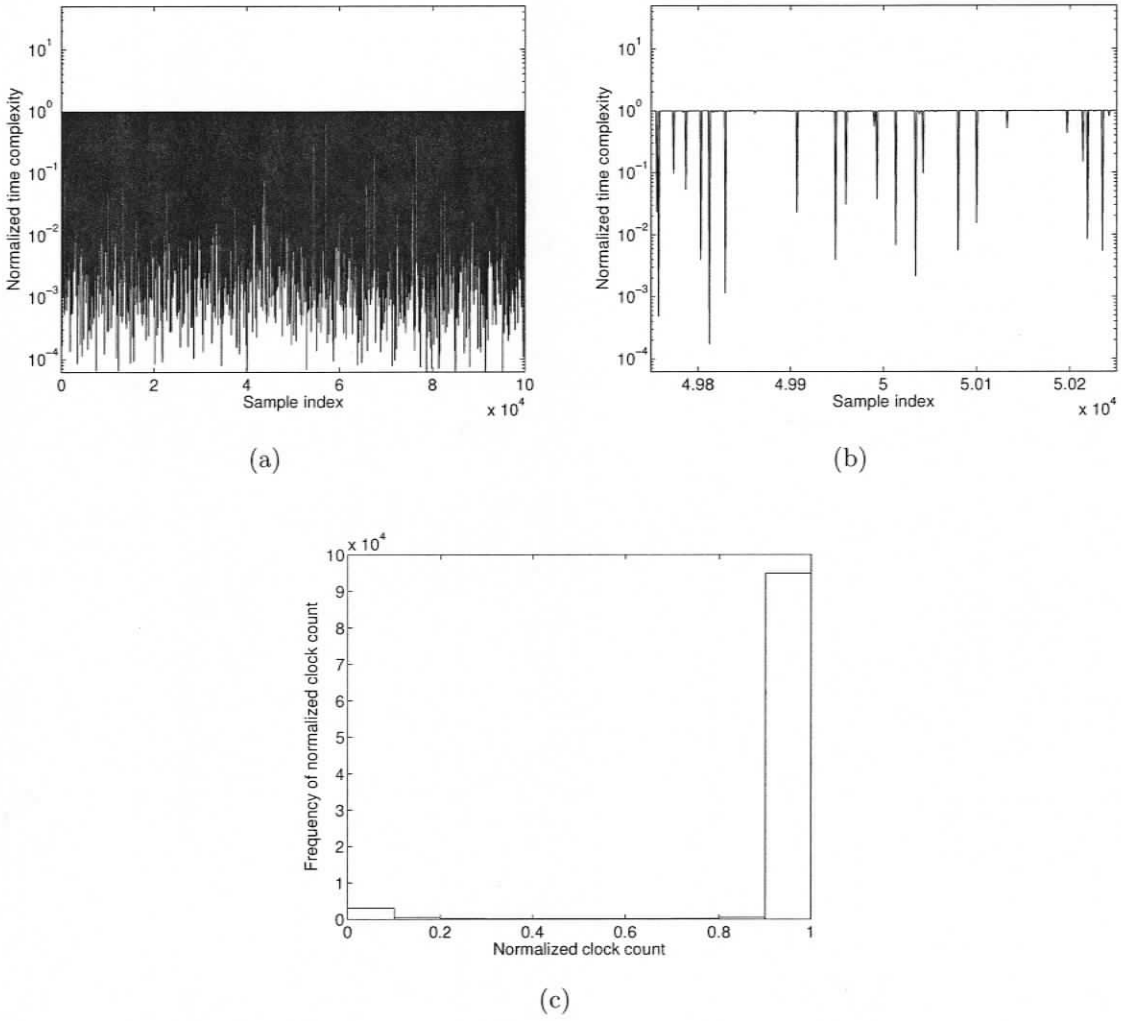


Figure 4.12: Experimental results for $\overline{m} = 10$: (a) for all data, (b) for 500 data, (c) the histogram.

This value is very promising in respect of a FPGA design.

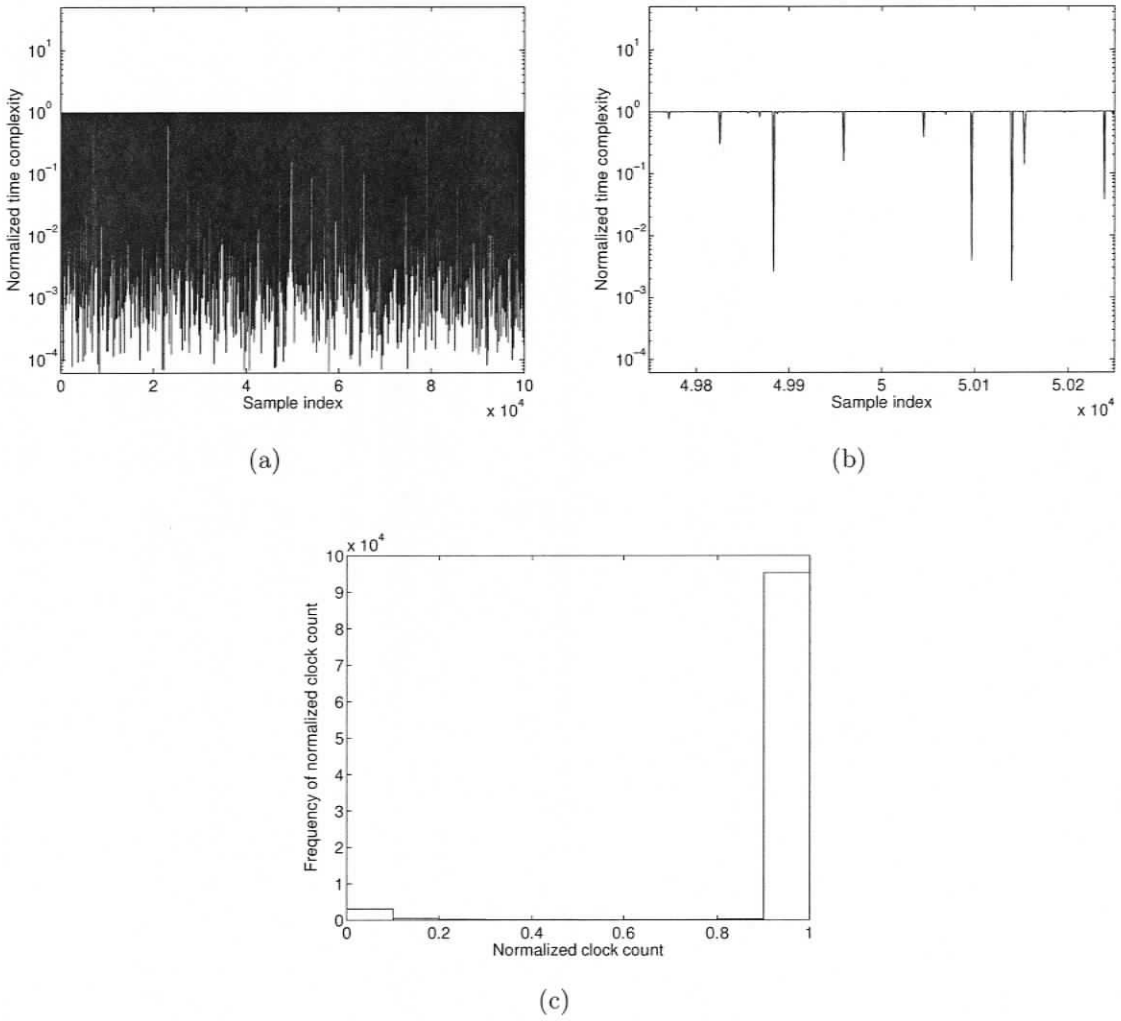


Figure 4.13: Experimental results for $\overline{m} = 20$: (a) for all data, (b) for 500 data, (c) the histogram.

4.7 Conclusion

In this chapter, we have proposed a novel embedding technique for processor array architectures for the string matching problem for the deep packet classification unit. Our technique has the following favorable characteristics:

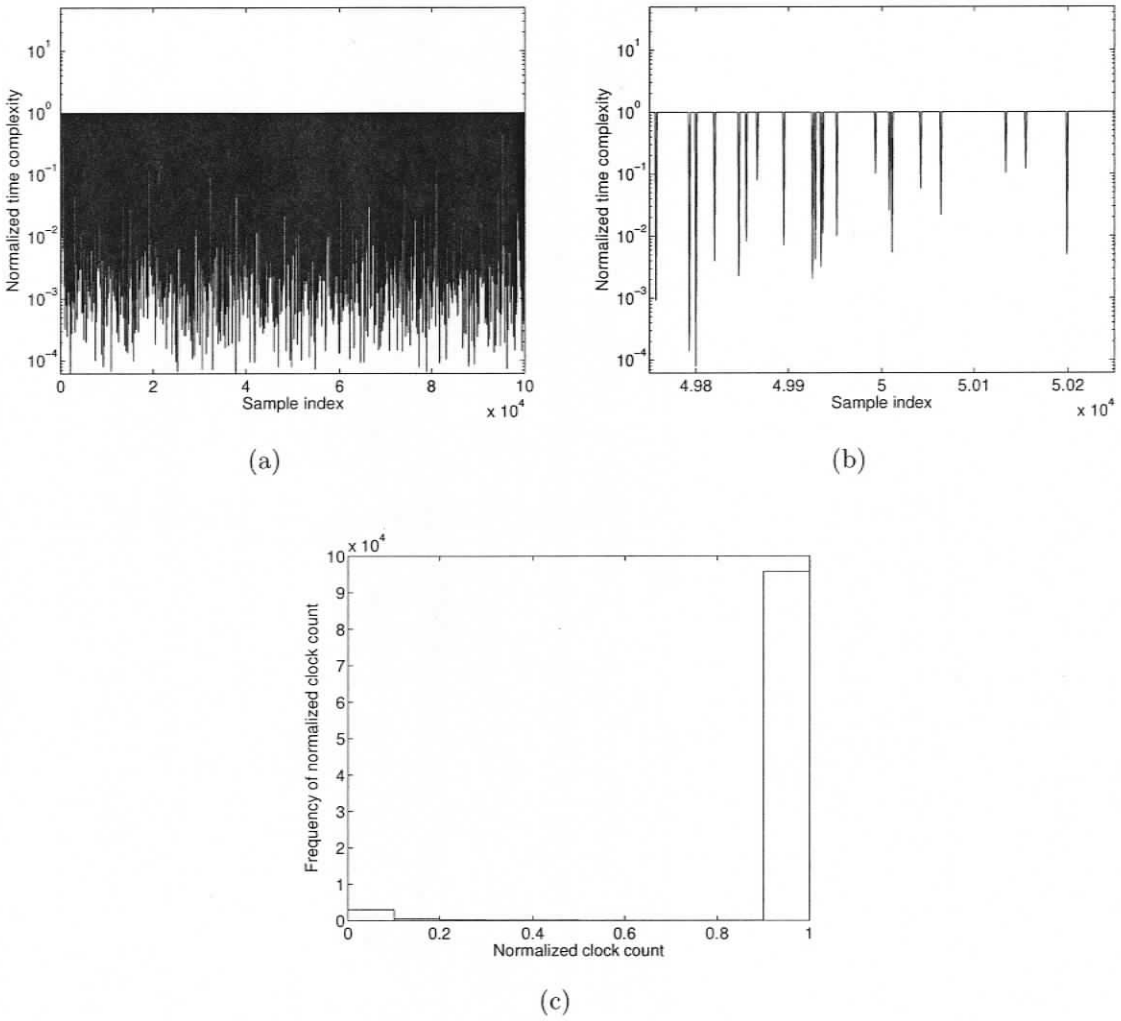


Figure 4.14: Experimental results for $\overline{m} = 40$: (a) for all data, (b) for 500 data, (c) the histogram.

- the proposed technique is simple to apply,
- interconnection patterns among PEs of the source and target arrays are same,
- execution times of the source and target processor arrays are same,

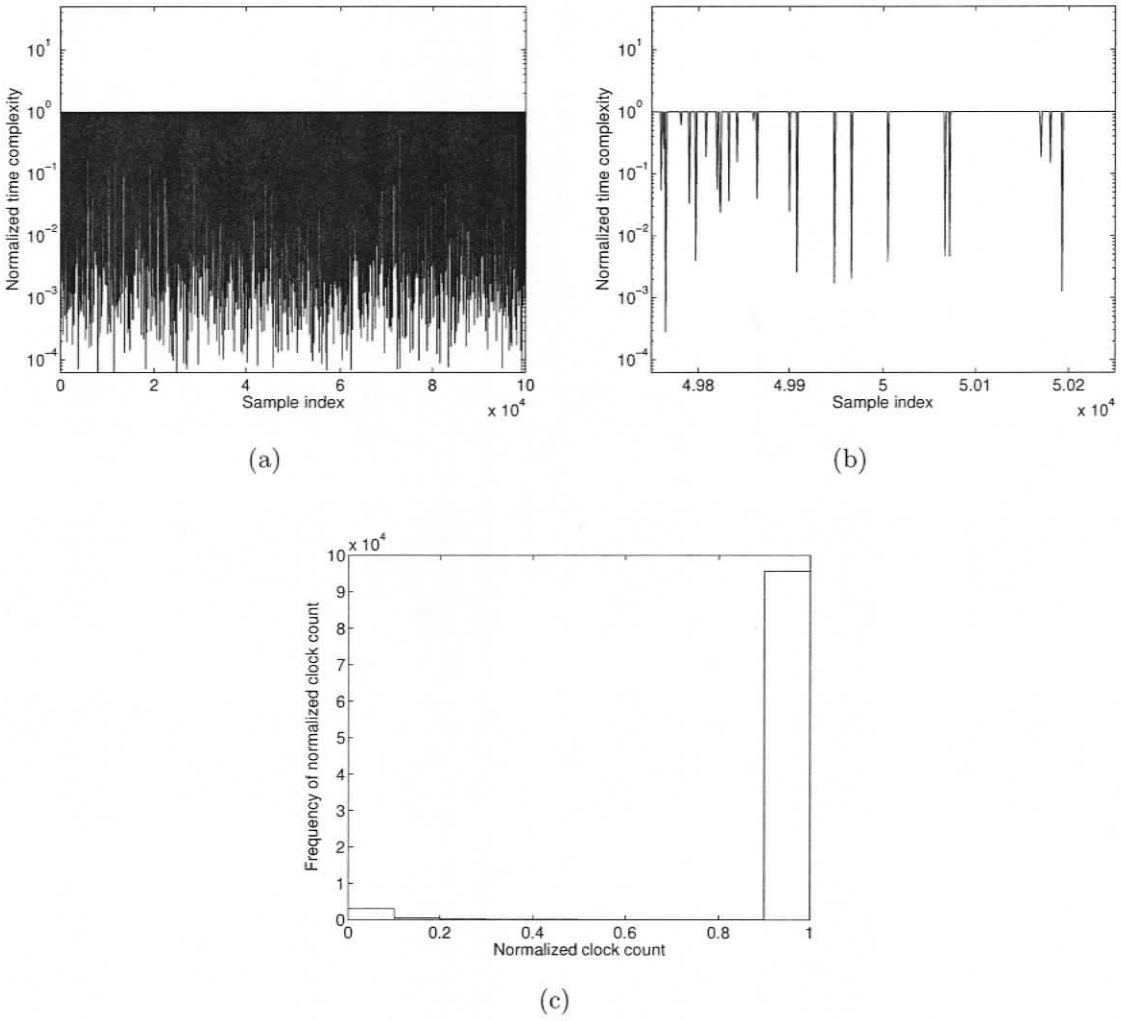


Figure 4.15: Experimental results for $\overline{m} = 50$: (a) for all data, (b) for 500 data, (c) the histogram.

- memory accesses of the source and target processor arrays are same, and
- I/Os of the target array are simple (no multiplexed I/O).

We have designed a Markov model for the embedded architecture. Through Markov analysis, we have proved that the performance of the embedded architecture shows

the same performance as the source processor array. We have checked the correctness of the operations and then synthesized the embedded architecture on Stratix II Altera FPGA. We have modeled the synthesized embedded array and done extensive numerical simulations in C language to support the Markov analysis results. In addition, from the synthesized results, we have shown the effects of the number of PEs on different Altera FPGA parameters. We have also calculated the throughput of our hardware. The performance of the embedded processor array is very good in terms of the calculated throughput on a FPGA.

Chapter 5

Longest Prefix Matching

In this chapter, we propose a novel variable-stride multi-bit trie data structure for IP-lookup table to assist fast IP-lookup and fast routing table update. This chapter is organized as follows. Section 5.1 gives the overview of the LPM problem. Section 5.2 summarizes some of the works on LPM. Section 5.3 picks the strides for our proposed multi-bit trie based system. Section 5.4 discusses our proposed data structure of the lookup table for LPM. Section 5.5 describes the technique to build our proposed data structure. Section 5.6 describes the technique to add an entry into the lookup table. Section 5.7 describes the technique to delete an entry from the lookup table. Section 5.8 describes the technique to search the best next hop, i.e., the technique for IP-lookup. Section 5.9 and Section 5.10 present the experimental results using our proposed scheme. Section 5.11 presents the mathematical analysis of our scheme. Finally, Section 5.12 concludes our chapter with future research direction.

5.1 Overview

The construction of lookup tables is the primary research goal for LPM, because lookup table dictates the performance of the IP-lookup, the memory requirements for lookup tables, and the performance of lookup table update. Generally there is a trade-off between time and space, especially lookup table update time and lookup table size. In this chapter, our goal is to make the time performance better at the expense of memory requirement. The memory requirement of our algorithm is bigger than some approaches, but it is well within the limit of current memory availability and price.

The trie [158] is a general purpose data structure for storing strings. Since the IP address of the IP-packet is a string of 0s and 1s, we can use the trie as a data-structure for the IP-lookup. But there are two problems with the trie structure: (i) for a big lookup table, the trie would be very big and (ii) more importantly, the number of the memory access would be large for a big trie. In fact, the number of the memory access depends on the depth of the trie structure and this depth is fixed for a particular protocol (e.g., 32 for IPv4). Since, the memory access has the major impact on the lookup time, we have to reduce the depth of the trie structure for faster lookup. The traditional technique to reduce the depth of the trie is to remove each internal node with only one child. This technique is known as path-compression and the path-compressed trie is sometimes referred as Patricia tree [159]. But this technique works well only for the sparsely populated trie. For the densely populated trie, another technique namely level compression is required. Thus Level-Compressed Patricia tree or LC-trie [44, 160–162] was introduced to reduce the depth of any trie structure. Although LC-trie shows better performance in terms of the number of nodes and the average depth of the trie, the path and level compressions makes it difficult (time consuming) to update the trie. But several millions updates are required per day

for a router [163]. For this reason, Pao, et al. proposed a technique that stores two additional information in each node of the LC-trie to make the update faster at the expense of the lookup time. For example, for Aads routing table, lookup rate is 2.0 million lookups per second for their technique, but 2.63 million lookups per second for the original LC-trie ([44]). While the original LC-trie can do few updates per second, but the modified technique by Pao, et al. can do thousands of updates per second.

In LC-trie, level compression depends on lookup table entries. Due to this dependency, LC-trie shows significant degradation in performance in some routing tables. In the worst case, the number of memory access would be 32 for IPv4. For this reason, researchers on LC-trie did direct lookup for the first 16-bits and do other modifications to reduce the dynamic nature of the LC-trie [44]. To reduce the dynamic nature of LC-trie, many researchers worked with fixed levels of the trie, called k-stride multi-bit trie [26, 164]. The depth of the subtrees combined to form a single multi-bit trie node is called the stride. To take advantage of the distribution of lengths of entries in a routing table, some researchers worked with variable-stride multi-bit trie. We have chosen variable-stride multi-bit trie as our data structure that proves the best in lookup and update performance than other approaches in our experiments. We have reviewed some more works on LPM in Section 5.2.

5.1.1 Problem of Expansion

Almost all researchers try to check first 16 bits (corresponds level 16 in the trie) of the prefix to make the search time faster [44]. If the prefix length is smaller than 16, then the prefix must be expanded to make the length = 16. The problems associated with this prefix-expansion is called Problem of Expansion. Figure 5.1 is drawn to explain the Problem of Expansion. For direct lookup, variable or fixed-stride multi-

bit trie structures, the Problem of Expansion must be taken care of. Each entry in the routing table has two fields: prefix and next hop. However, from now on, for each of description, each entry will be described by a tuple of three fields: (prefix, prefix length, next hop), where prefix is represented in hexadecimal format. We use the following conventions while drawing the trie: ● means a node that corresponds to a routing entry, ◉ means an internal node, and ○ means a leaf node.

In Figure 5.1, nodes n_1, n_2, and n_3 holds the values for entries, e_1, e_2, and e_3, respectively, where e_1 = (8000 0000, 15, 2), e_2 = (8002 0000, 14, 3), and e_3 = (8000 0000, 14, 4). The Problem of Expansion may occur in two ways as explained below.

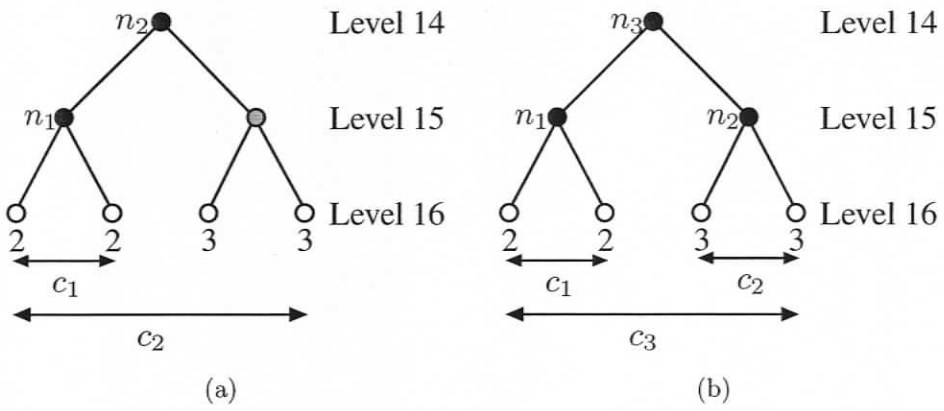


Figure 5.1: The Problem of Expansion: (a) partial confusion, (b) complete confusion.

- **Partial Confusion:** In Figure 5.1(a), e_1 and e_2 are extended by 1 and 2 bits respectively. e_1 covers the address range c_1, while e_2 covers the address range c_2. Since e_1 has the larger prefix length, next hops for the address range c_1 are 2. The search or lookup algorithm may be confused, if this Problem of Expansion is not taken care of.

- **Complete Confusion:** This case is more complex than the first one. There

is no place for e_3 in the leaves. For correct lookup, this Problem of Expansion must be taken care of.

Without the consideration of above two problems, the LPM technique may not work on all routing tables. But all of the existing approaches ignored these problems or did not explicitly solve them. In addition to solve these problems, our goal is to reduce the overall memory access, not only lookup table access during IP lookup and lookup table update.

5.2 Existing Works

In this section, we discuss on some techniques on LPM. Discussion on more techniques on LPM can be found in [165].

Degermark, et al. used variable-stride multi-bit trie for lookup table [164]. Their goals were to minimize memory requirement and lookup time. For example, their technique requires 160 kB for 32,732 entries. In their simulation environment, the lookup speed is 2.2 million lookups per second. Since they do not keep the information after changing the prefix during table build-up, it makes the deletion of an entry very time consuming.

Sangireddy et al. varied patricia trie [159] to propose two algorithms: Elevator-Stairs algorithm and $\log W$-Elevators algorithm [166]. Like works in [164], their main goal is to reduce lookup time and memory requirement. According to their simulation results, Elevator-Stairs and $\log W$-Elevators algorithms lookup 15.08 million lookups per second and 20.5 million lookups per second, respectively. These two algorithms requires 517 kB and 1413 kB, respectively, for 33796 entries of Aads routing table. But their approaches are time-consuming to update the routing table like patricia tree.

Srinivasan et al. used variable-stride multi-bit trie for lookup table [167]. The prefix expansion in their technique occurs based on the optimal memory requirements. Reference [168] improved the algorithm used in [167]. From the simulation results, search time for the technique in [168] is 2 to 17 times faster than the technique in [167]. But the problem is that when prefix expansion occurs, some information is lost which makes the updating very costly.

Gupta et al. used almost the same technique as ours [169]. Their technique requires single memory access per lookup. But the problem is that when prefix expansion occurs, some information is lost. This results in long update time of their technique .

Reznik studied adaptive multi-digit branching of tries in [170]. His approach has the same problem as the work in [44].

Ioannidis et al. formulated the level-compression problem as a variation of the knapsack problem to reduce the number of memory access during lookup [171]. However, their technique possess the same problem as the original LC-trie work described in [44]. But technique in [171] takes 9/22 less memory access during lookup than that using the technique in [44].

Lu et al. proposed two partitioning techniques which can be applied to fixed-stride tries to improve the lookup performance [172]. But the loss of information during partitioning increases the update overhead.

Taylor et al. employed parallel IP lookup engines for fast lookup and fast update [173]. The engines are based on Tree bitmap algorithm that compares 4-bits at a time. However, the 4-bit stride may not always produce best lookup time and memory requirement.

Ioannidis et al.used directed acyclic graph to optimize the the level compression of the LC-trie in [174]. This optimization helps 34% increase in level compression and 1.21% increase in memory requirements than those of LC-trie [44]. Their technique

requires used additional data structures to store extra information than those required in LC-trie. Update time is very time consuming using their technique. The update time reported their paper is $\mathcal{O}(n)$, where n is the total number of entries in the trie..

Kaxiras et al. proposed a set-associative memory structure for the variable-stride multi-bit trie [175]. Their technique have also pruned some of the trie-nodes to reduce the memory requirement like [176]. This loss of information makes it very difficult for updating the lookup table.

Lampson et al. used the binary search on the lookup table for the LPM problem [47]. Updating the lookup table is the main problem of this technique. To make the search time faster, their technique uses direct lookup in the first 16-bits, but ignored the problems as shown in Figure 5.1.

Sundstrom et al. proposed block tree based variable stride trie for lookup table structure [177]. Block tree is similar to the multi-way binary search used in [47]. Their technique's lookup performance is much higher than the technique described in [47].

Dharmapurikar used Bloom filters to determine all prefix matches and store results in a W-bit ($W = 32$ for IPv4) vectors [178]. Lookup table is built in a hash table which is searched based on the bit-vector. Obviously, one hash-table access is required for the IP-lookup. However, due to false positive, multiple accesses may be required in few cases. The bottleneck of their problem is the usage of the bit-vector. Also their technique uses direct lookup to reduce the average number of the hash-table access, but ignored the problems as shown in Figure 5.1. Also update is costly using their technique.

Several authors proposed CAM-based solution for LPM problem [176, 179–182]. But CAM-based solutions has the inherent disadvantages compared to others in terms of cost and power consumption [178].

From the above discussion, almost all researchers looked for fast lookup time

only. One work using multiple Parallel search engines has good performance on both lookup and update time [173]. However, based on simulation results, our work shows better performance in both lookup and update time. In addition to that this chapter first explicitly elaborates the solution of the problems portrayed in Figure 5.1.

5.3 Selection of Strides

We use variable-stride multi-bit trie for the LPM problem. In this section, we select the strides to form the trie for lookup table. Our selection process depends on the distribution of the entries in the routing tables. At this time, we discuss for IPv4 packets. Later in this chapter, we will explain how to adapt our approach for IPv6 packets.

To do our experiments, we have collected 11 routing tables available in [183]. The properties of these tables are summarized in Table 5.1. To save space, we have drawn the distribution of prefix lengths only for three routers (Paix, Att, and Oregon) in Figure 5.2. Other routing tables have similar distributions. From Table 5.1 and Figure 5.2, prefix lengths of most routing entries are between 16 to 24. For this reason, we have divided the prefix lengths into three divisions: (i) 1 to 16, (ii) 17 to 24, and (iii) 25 to 32. There is another reason for picking up these three values. Since 16, $24 - 16$, $32 - 24$ are power of 2, divide the prefix length into these three is easier to implement. We name this as $scheme_1$. In another scheme, $scheme_2$, we divide the IPv4's 32 bits into two divisions: (i) 1 to 24, and (ii) 25 to 32. In the third scheme, $scheme_3$, we use only one level of 32 bits. Although the lower number of levels reduces the number of lookup table access, they increase the memory requirements dramatically. Also the average time requirement of the lookup is not as advantageous as the same proportion of the lookup table access counts.

We discuss only $scheme_1$, since the discuss is applicable to $scheme_2$ and $scheme_3$.

Table 5.1: Properties of the routing tables. PL means Prefix Length.

Site	Routing entries	Next-hops	PL<=16 (%)	16<PL<=24 (%)	PL>24 (%)
Aads	32,505	38	8.05	86.61	5.33
Att	121,711	26	6.96	93.02	0.02
East Attcanada	127,561	92	6.74	89.78	3.48
Funet	41,328	18	15.07	84.86	0.07
Mae West	71,319	38	7.08	90.55	2.37
Oregon	142,883	67	6.03	90.18	3.79
Pacbell	45,184	1	7.29	88.86	3.85
Paix	17,766	28	7.11	83.49	9.39
Telstra	104,096	182	7.37	88.68	3.96
Telus	126,687	76	6.75	90.99	2.25
West Attcanada	127,576	92	6.72	89.73	3.55

5.4 Structure of the Trie

In $scheme_1$, the trie has three levels represented by three arrays: *Level*16, *Level*24, and *Level*32. In our approach, *Level*16 holds all 2^{16} nodes available at level 16 in a trie as shown in Figure 5.3(a). Now let there be a prefix of prefix length 20. The prefix is associated with one of the nodes in *Level*16. That node in *Level*16 now has 2^8 children stored in *Level*24 as shown in Figure 5.3(b). Similarly, if there is a prefix of length 30, then the associated node in *Level*16 has 2^8 children stored in *Level*24 and the associated node in *Level*24 has 2^8 children stored in *Level*32 as shown in Figure 5.3(c). Thus the size of *Level*16 is 2^16 (fixed). But the of *Level*24 and *Level*32 depends on the entries in the routing table. Although our proposed trie

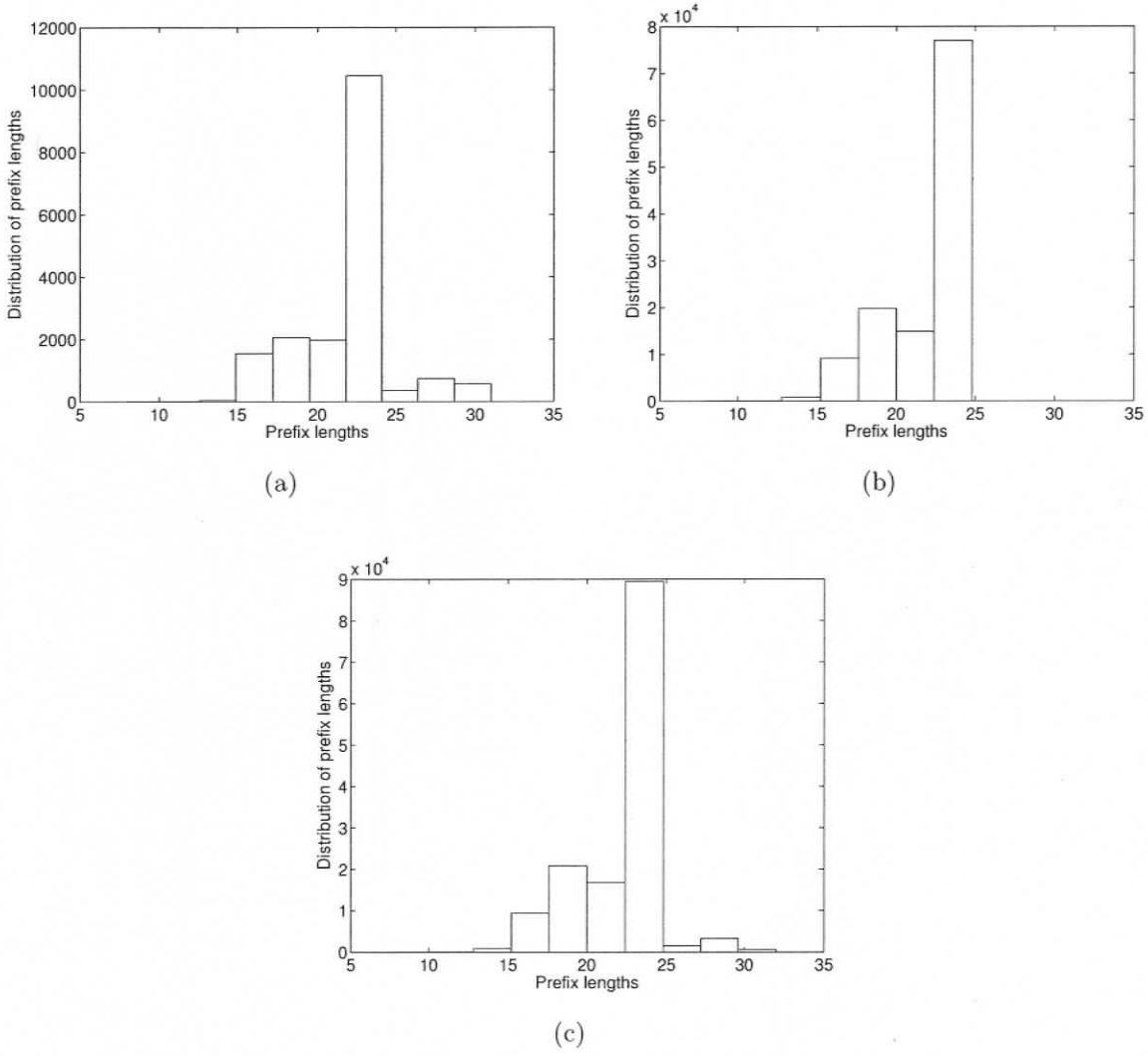


Figure 5.2: Distribution of the prefix length for Paix router in (a), Att router in (b), and Oregon router in (c).

organization increases the memory requirements, it improves the update time of the lookup table significantly.

In our scheme, each node of the three levels stores the following:

- *Next_hop*: Unlike others [26], we store next-hop in each node for fast lookup.

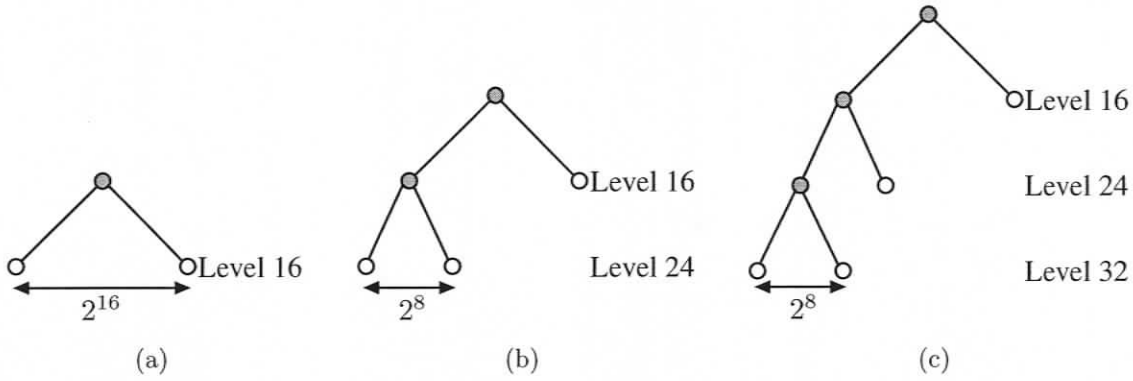


Figure 5.3: Organization of the trie for level 16 in (a), level 24 in (b), and level 32 in (c).

Although it increases the memory requirements, it increases the lookup performance significantly. From Table 5.1, there are maximum 182 next-hops in a routing table. So, 8 bits are sufficient for *Next_hop*.

- *Address*: each node has to store the address of the left-most child of the node. To keep the address field smaller, we store only the base address that must be multiplied by 2^8 (equivalently 8-bit left shift which is faster than multiplication). From our simulation results, 15 bits are enough for the *Address* field of *Level*16 and *Level*24. For *Level*32, no address field is required, but we keep 1-bit flag to identify valid and invalid leaf.

- *Identification*: this 1-bit flag is to identify the starting node covered by a prefix. This is required to update the lookup table (trie structure) as described in Section 5.6 and Section 5.7.

- *Cover*: if the node is covered by a prefix entry, then cover stores the distance between the level of the node and the prefix length. In Figure 5.1(a), n_1 (i.e. e_1) and n_2 (i.e. e_2) cover two and four nodes in level 16, respectively. However, to

reduce the Cover field, we store the value $\log_2(cover)$ in the Cover field. Thus, 4-bit is required for *Level*16 and 3-bit is required for *Level*24 and *Level*32.

- *Duplicate*: this 1-bit flag indicates that the node is covered by more than one prefix entry.

Thus only 4 bytes are required for each node in *Level*16 and *Level*24, while *Level*32 requires only 2 bytes for each node. We have drawn a node in the trie in Figure 5.4 showing all fields. For example, the left-most leaf node of the trie in Figure 5.1(a)

Duplicate	*Cover*	*Identification*	*Address*	*Next_hop*

Figure 5.4: Node of the trie of our proposed scheme.

looks like Figure 5.5. The right-most leaf node of the trie in Figure 5.1(a) looks like

TRUE	1	TRUE	0	2

Figure 5.5: Left-most leaf node of the trie in Figure 5.1(a).

Figure 5.6.

FALSE	2	FALSE	0	3

Figure 5.6: Right-most leaf node of the trie in Figure 5.1(a).

To overcome the problems depicted in Figure 5.1, we have introduced another array, *Level_CQX* (X = Level number) to store the next-hops for the node, n that has *Duplicate* = TRUE. In this case, *next_hop* contains the address of *Level_CQX*. For this reason, the size of *next_hop* is more than 8-bit in actual implementation.

From our simulation, it is seen that 11 bits for *Level*16, 12 bits for *Level*24, and 10 bits for *Level*32 suffice for the implementation. Even though the size of *next_hop* has been increased, the overall size of each node remains the same as calculated above.

Each element of *Level_CQX* consists of: *Next_hop* and *Value*. *next_hop* of *Level_CQX* contains next_hop of an entry e at index i, if e corresponds to n and *cover* of e is $i = X - PL$, where PL is the prefix length of e. This type of assignment of next_hop helps for fast lookup (see Section 5.8). Thus the size of *Level_CQ*16 is 16, while the size of *Level_CQ*24 and *Level_CQ*32 is 8 bit only.. *Value* is 1-bit boolean type field that is TRUE if the *next_hop* contains the last entry from the same e, otherwise it is FALSE. For example, the contents of the array for the left-most node of Figure 5.1(a) are given in Figure 5.7.

	NH	Val
0	2	FALSE
1	2	TRUE
2	3	TRUE
	⋮	

Figure 5.7: Array for the left-most node of Figure 5.1(a). NH = Next-hop, Val = Value, and D = default next-hop.

Thus our proposed trie structure requires six arrays: *Level*16, *Level*24, *Level*32, *Level_CQ*16, *Level_CQ*24, and *Level_CQ*32. Out of these six arrays, *Level*16 is fixed in size and its size does not change after addition or deletion of an entry from the routing table (trie). The sizes of other five arrays may increase or decrease when a new entry is added or deleted from the trie. For better utilization of the memory, we use circular queue [184] to store the available or free addresses for these arrays. To add or store an element in one of these arrays, we have to get an address from the

associated queue. To delete an element, we have to put the index in the associated queue so that the index can be used for another entry.

5.5 Building the Trie

To build the trie from a lookup table, we have to initialize all queues with the free memory locations of the associated arrays. We have also to initialize $Level\_CQ16$, $Level\_24$, and $Level\_CQ32$ with 0 and the fields of each node of the array for the smallest trie-level, i.e., $Level16$ with the values as shown in Figure 5.8. $Leaf\_node$ is

FALSE	0	FALSE	$Leaf\_node$	$Default\_next\_hop$

Figure 5.8: Initial values of $level16$.

the predefined address to indicate the leaf node of the trie and $Default\_next\_hop$ is the user-defined default next_hop.

After these initializations, we use addition (Section 5.6) to add all the entries of the lookup table in the trie.

5.6 Addition

Addition is required to build the trie and add a new entry into the trie (i.e. update lookup table). Assume we have to add $e = (P, PL, 5)$. We describe here only how $Level16$ and $Level\_CQ16$ are affected by the addition of e. Same description will apply for $Level24$, $Level\_CQ24$, $Level32$ and $Level\_CQ32$. From now on, for ease of discussion, assume that first 16 bits of the prefix is represented by P_1, second 8 bits of the prefix is represented by is P_2, and last 8 bits of the prefix is represented by P_3.

We have divided our works into three subsections based on values of PL as described below:

5.6.1 $PL = 16$

Two situations may occur when $PL = 16$:

- Assume $Level16$ at P_1 is not covered by any entry, i.e., $next\_hop$ field of $Level16$ at P_1 contains $Default\_NH$ as shown in Figure 5.9(a). After entering e, the trie would look like Figure 5.9(b) and node n would look like Figure 5.10. Here we ensure that if there are two or more entries with same P and same PL, the last one will be added into the trie. $identification$ must be TRUE because it is now the starting node for P. $duplicate$ field retains its value as FALSE since this is the first entry in that node. $Address$ field is not required to be changed here.

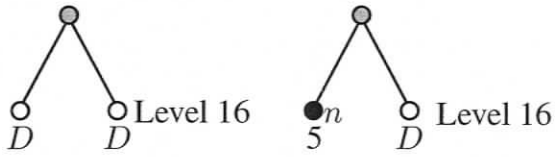
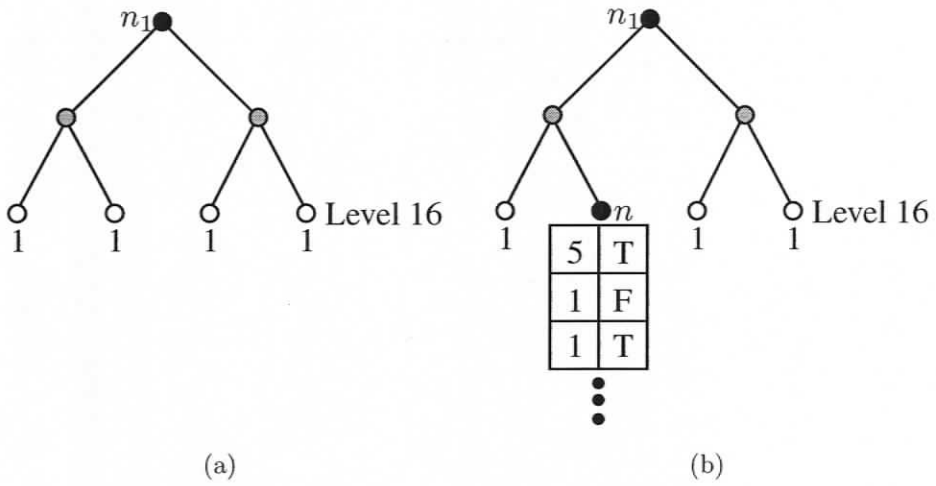


Figure 5.9: Addition of e where $PL = 16$ and $Level16$ at P_1 is not covered by any entry: (a) trie before addition and (b) trie after addition. D means $Default\_NH$.

FALSE	0	TRUE	NC	5

Figure 5.10: Contents of node n of Figure 5.9(b). NC means no change.

- Assume *Level*16 at P_1 is covered by an entry $e_1 = (P, 14, 1)$ as shown in Figure 5.11(a). The *duplicate* field will become TRUE and new array *Level_CQ*16 is required. *cover* becomes invalid, *identification* becomes TRUE, *address* remains unchanged and *next_hop* contains the address of *Level_CQ*16.



(a) (b)

Figure 5.11: Addition of e where $PL = 16$ and *Level*16 at P_1 is covered by an entry: (a) trie before addition and (b) trie after addition.

5.6.2 $PL < 16$

Assume we are going to add $e = (P, 13, 5)$. There are already four entries, e_1, e_2, and e_3 in its cover as indicated by n_1, n_2, n_3, and n_4 in Figure 5.12(a). Adding e starts with changing the *Level_CQ*16 at node n_5. Since *cover* of e is 3 which is grater than that of e_2 (= 2), add process jumps 2^2 nodes ahead to n_6, where *Level_CQ*16 is changed accordingly. In n_6, the *cover* of e_3 is 1 which is smaller than the that of e, so again, the add process jumps 2^1 nodes ahead to n_4. After changing *Level_CQ*16 at n_4, the add process of e terminates in the last node of its cover at n_7, where *next_hop* becomes 5.

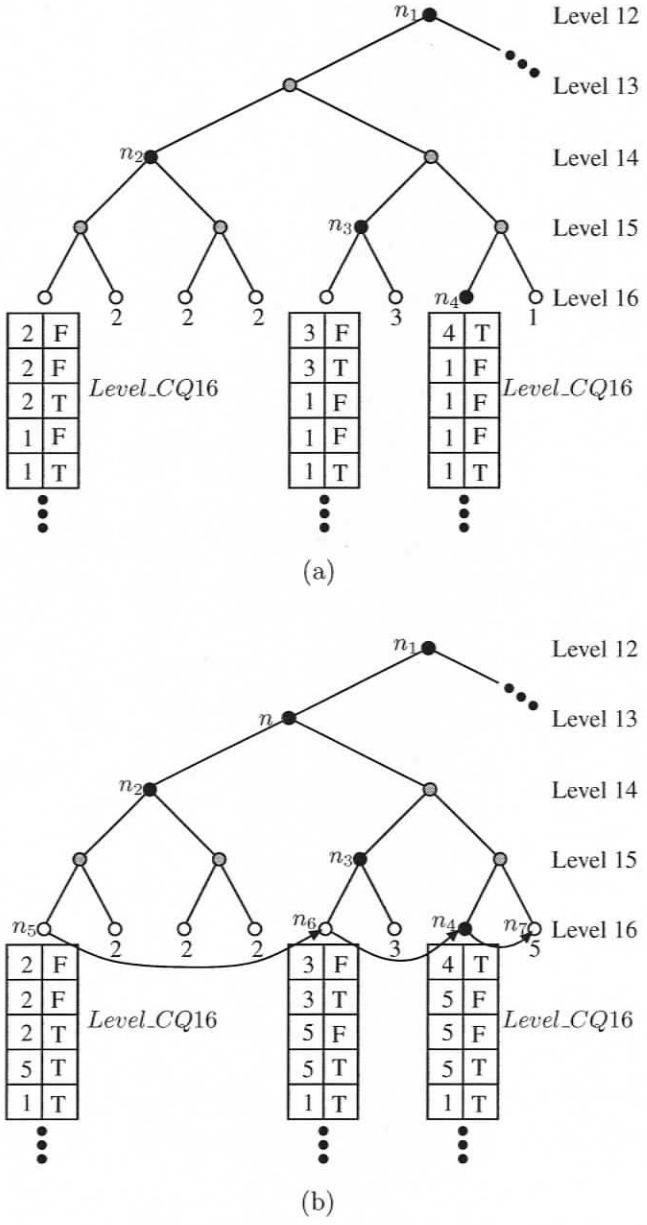


Figure 5.12: Addition of e where $PL < 16$: (a) trie before addition and (b) trie after addition.

5.6.3 $PL > 16$

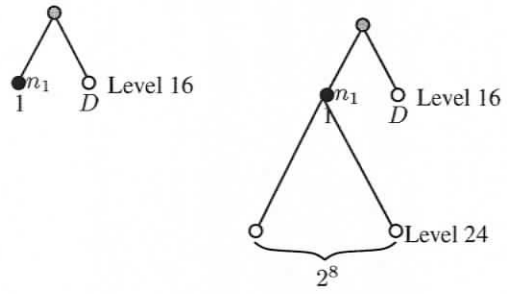


Figure 5.13: Addition of e where $PL > 16$: (a) trie before addition and (b) trie after addition.

Assume $PL = 23$ and P_1 location of $Level16$ is already covered by e_1 as shown in Figure 5.13(a). If the *address* of n_1 is not $Leaf\_node$, then the same procedure is applied as described above. If the *address* of n_1 is $Leaf\_node$, then we have to add 2^8 children of n_1 and the *address* of n_1 is the address of the leftmost children in $Level24$. All the children are also initialized as the leaf node as described in Section 5.5. After the initialization, $Level24$ and $Level\_CQ24$ will be changed according to the same procedure as described above.

5.7 Deletion

To delete an entry e, we have to find the next-hop that will take place in the nodes covered by e. To delete an entry in the lookup table (trie), we have to consider two situations:

- **Case 1:** The *duplicate* field of the leftmost covered node in the next closest level is TRUE, i.e. the leftmost node is covered by multiple entries.

- **Case 2:** The *duplicate* field of the leftmost covered node in the next closest level is FALSE, i.e. the leftmost node is covered only by the entry to be deleted.

Each case is dealt with two phases:

- find the next-hop to be replaced in the covered node, and
- change all nodes covered by the entry to be deleted.

The above two cases will be described in the following two subsections.

5.7.1 Case 1

Finding the next-hop to be replaced is very easy in case 1. Assume we have a lookup table of three entries: e_1 = (80000000, 15, 2), e_2 = (80000000, 14, 3), and e_3 = (80000000, 13, 4). The portion of the trie wold look like Figure 5.14(a), where nodes n_1, n_2, and n_3 hold the entries e_1, e_2, and e_3, respectively. Also assume that we

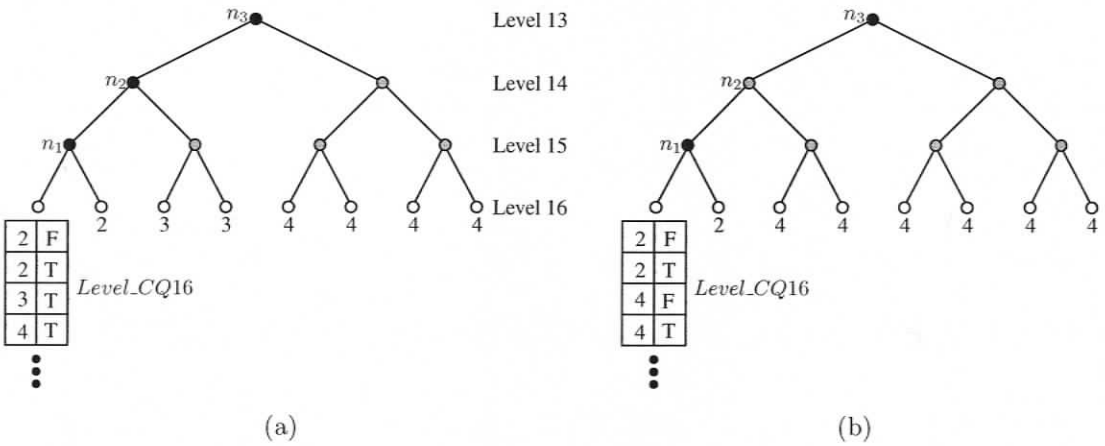


Figure 5.14: Deleting an entry from the lookup table: (a) shows a portion of the trie before the deletion and (b) show the trie after the deletion of e_2.

want to delete entry e_2. Since the cover of e_2 is 2 $(16 - PL)$, we have to search the

*Level_CQ*16 from index 3 (*cover* + 1) for the next-hop to be replaced with. In our example, it is 4 as shown in Figure 5.14. So, *Next_hop* and *Value* of *Level_CQ*16[3] are replaced with 4 and FALSE, respectively. However, if the *Level_CQX* holds only two next-hops as shown in Figure 5.15(a), then after deleting e_2, the *duplicate*

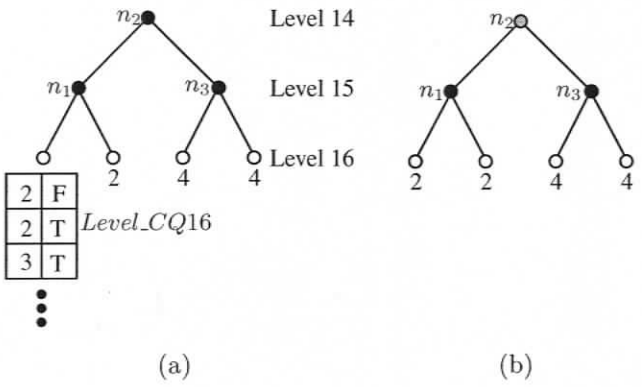


Figure 5.15: Deleting an entry from the lookup table: (a) shows a portion of the trie before the deletion and (b) show the trie after the deletion of e_2.

field of the left-most leaf node becomes FALSE and the *next_hop* field of the left-most leaf node holds 2 (= *next_hop*) instead of the address of *Level_CQX*, because array *Level_CQX* is no longer required. In addition to changes in *Level_CQX*, the *_hop* fields of the covered nodes are to to changed as shown in Figure 5.14(b) and Figure 5.15(b).

5.7.2 Case 2

Finding next-hop to be replaced with in this case is little bit of tricky. We have to start finding next-hop from the previous location (*l_addr*) of the left-most node covered by the entry to be deleted. If the node in *l_addr* holds $default\_NH$ as the next-hop, then we have already found the next-hop and it is $default\_NH$. If the node

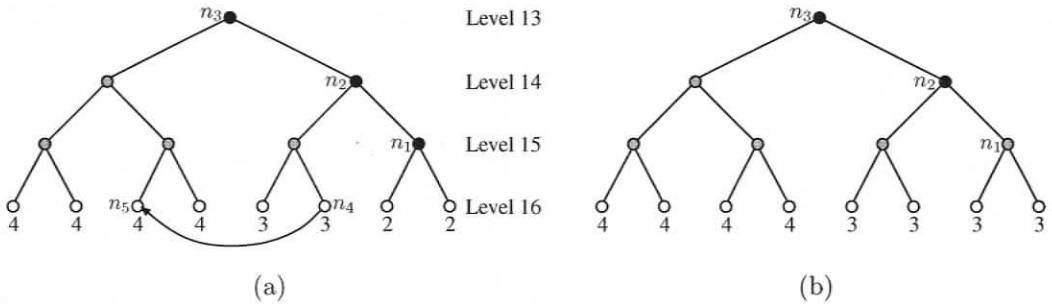


Figure 5.16: Deleting an entry from the lookup table: (a) shows a portion of the trie before the deletion and (b) show the trie after the deletion of e_2.

at l_addr does node hold $default\_NH$ as the next-hop, then we recalculate l_addr as,

$$l\_addr = l\_addr - 2^{cover} + 1$$

where cover is the value of cover for the node at l_addr. Before recalculating l_addr, we keep the previous value of l_addr in t_addr. If the identification of the node at new l_addr is FALSE, then next-hop equals to next-hop of the node at t_addr as shown in Figure 5.16. However, if the identification of the node at new l_addr is TRUE, we have to check whether the cover of the new node equals to the cover we have used to recalculate l_addr. This checking is required to determine whether the new node is within the cover of the node we are looking for. If the cover of the new node equals to the cover we have used to recalculate l_addr, it is within the cover of the node we are looking for as shown in Figure 5.17(a) (n_5) and Figure 5.18(a) (n_5). In this case, we have to check the *duplicate* field of the new node. If it is TRUE as in Figure 5.17(a) (n_5), we have to calculate two distances, d_1 and d_2 for all covers in $level\_CQX$ as,

d_1: distance between the new node and the left-most node covered by the entry to be deleted.

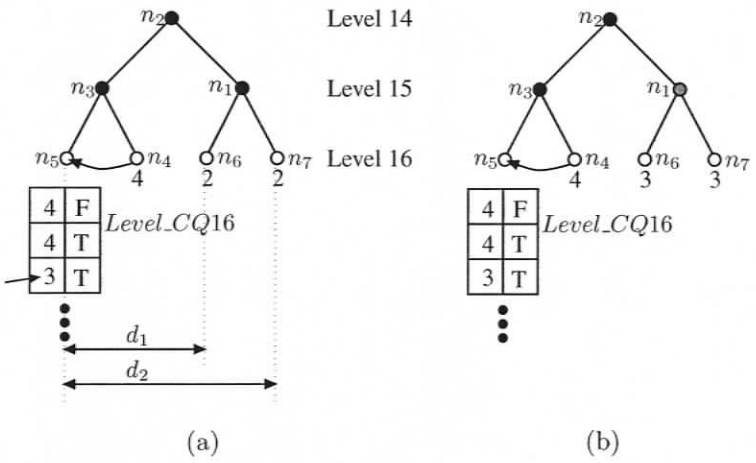


Figure 5.17: Deleting an entry from the lookup table: (a) shows a portion of the trie before the deletion and (b) show the trie after the deletion of e_2.

d_2: 2^{cover}.

We continue to search from lower to higher cover until we find a cover for which $d_2 \geq d_1$. This cover corresponds to the next-hop we are looking for. If there is no such cover, we have to recalculate *l_addr* by

$$l\_addr = l\_addr - 1$$

and then continue searching next-hop using the above mentioned method.

However, if the *duplicate* is FALSE as in Figure 5.18(a) (n_5), we have to continue to search next-hop based on above discussion. In Figure 5.18(a), as the searching continue, *l_addr* goes to n_7 that does not pass the cover test. So the next-hop= 4 as got in n_6 at the previous location of *l_addr*.

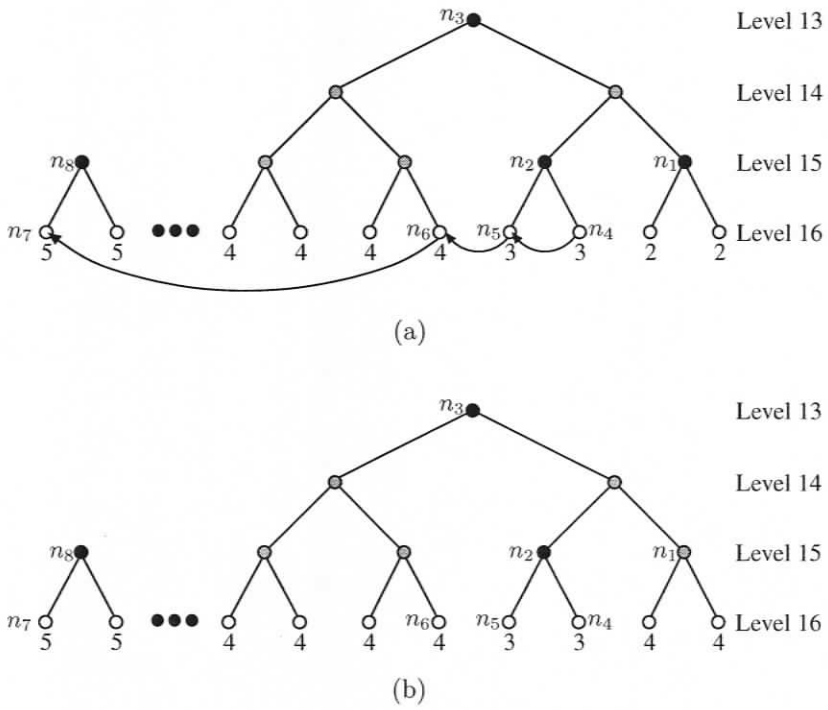


Figure 5.18: Deleting an entry from the lookup table: (a) shows a portion of the trie before the deletion and (b) show the trie after the deletion of e_2.

5.8 Search

We keep the search process straight-forward to make it faster. If the $PL < 16$, then we find the next-hop from $Level16$ or $Level\_CQ16$. If the *duplicate* of $Level16[P_1]$ is 1, the next-hop would be equal to $Level\_CQ16[cover]$, otherwise next-hop would be equal to the *next_hop* of $Level16[P_1]$, where $cover = 16 - PL$. If $16 < PL \leq 24$, *address* is calculated by,

$$Level16[P_1].address << 8 + P_2$$

, where $<< 8$ means 8-bit left shift or multiplied by 2^8. Since multiplication take longer time, we use faster shift operation. Then next-hop would be equal to $Level24[address].next$

and *Level_CQ*24[*cover*], if duplicate is 0 and 1, respectively. Similarly, we calculate the next-hop for the entries with $PL > 24$ from *level*32 or *level_CQ*32.

The pseudo-code of the search algorithm is available in [85].

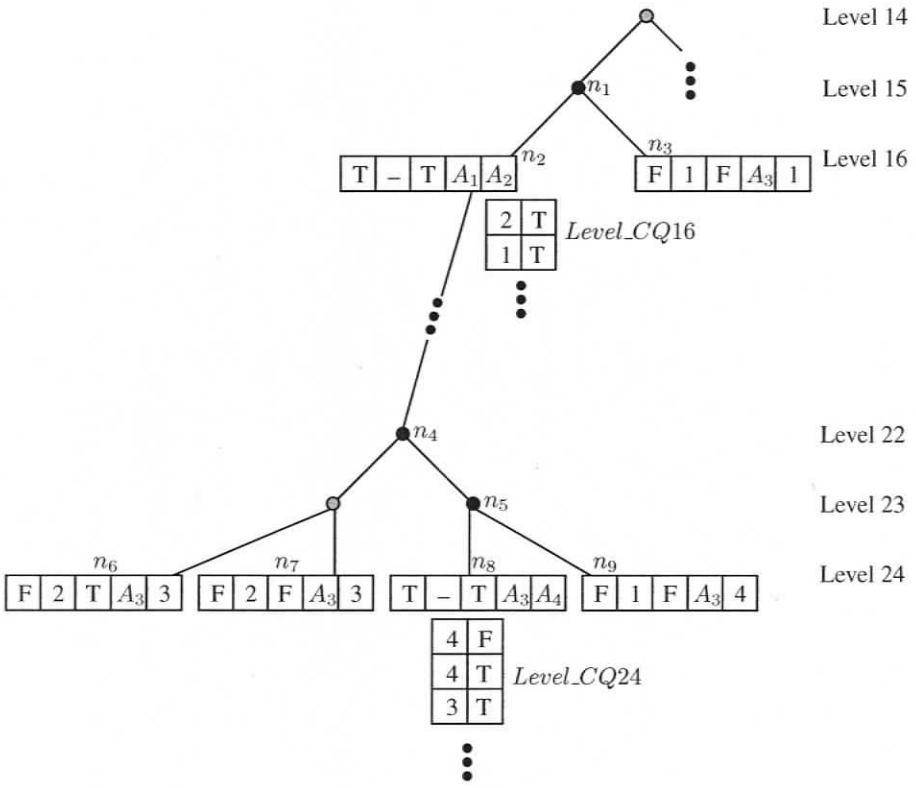


Figure 5.19: Trie structure for e_1 = (60000000, 15, 1), e_2 = (60000000, 16, 2), e_3 = (60000000, 22, 3), and e_4 = (60000200, 23, 4). A_3 is the pre-defined value for leaf-node.

For example, assume we have a lookup table with four entries: e_1 = (60000000, 15, 1), e_2 = (60000000, 16, 2), e_3 = (60000000, 22, 3), and e_4 = (60000200, 23, 4). The trie structure of the lookup table looks like Figure 5.19, where e_1, e_2, e_3, and e_4 corresponds to nodes n_1, n_2, n_4, and n_5, respectively. We are going to search the next_hop, x for e = (60000200, 24, x).

Here $P_1 = 6000$. Since $PL = 24 > 16$, we have to get address from *address* of *Level*16[6000]. From Figure 5.19, we get this address as A_1, which is the left-most child of n_2. Now, $P_2 = 02$. Then we have to search x in *Level*24[$02 + A_1$] which is noted by n_8 in Figure 5.19. But *duplication* of *Level*24[$02+A_1$] is TRUE, that means, we have to search x in *Level_CQ*24, whose address is in *next_hop* of *Level*24[$02+A_1$]. The cover of e is $24 - PL = 0$. So, x is the next-hop in *Level_CQ*24[0], i.e., 4.

5.9 Experiment Using $Scheme_1$

We have written the proposed scheme in C language and compiled using `gcc` 3.2.2 compiler. The machine we have used has the following configurations: Intel Celeron 1.8 GHz, data bus 400 MHz, cache memory (type L2) 256 kB, RAM 256 MB (DDR SDRAM – 266 MHz). We have used routing tables of 11 sites available in [183].

5.9.1 Experiment on Memory Requirements

Memory requirements of the 11 routing tables are summarized in Table 5.2. *Level*16 requires same memory for all routing tables, because *Level*16 has static number of nodes and it is 2^{16}. The size of each node is 4 bytes. Thus the total memory requirement for *Level*16 is 262,144 bytes. On the other hand, *Level*24, *Level*32, and *Level_CQX* have a dynamic number of nodes based on the nature and distribution of the routing entries. However, according to the routing entries' distributions given in Table 5.1, *Level*24 and *Level_CQ*24 should be largest among *LevelX* and *Level_CQX*, respectively, as it is reflected in Table 5.2.

Memory requirements of the routing tables in Table 5.2 are presented in Table 5.3 as a percentage of the total memory requirement.

Generally, larger routing tables require larger memory. Since in our technique,

Table 5.2: Memory requirements (Bytes) of the routing tables.

Site	L16	L24	L32	L_CQ16	L_CQ24	L_CQ32	Total
Aads	262,144	3,924,992	221,696	144	4,152	24	4,413,152
Att	262,144	7,541,760	10,240	1,424	25,624	0	7,841,192
East Attcanada	262,144	7,557,120	281,600	1,472	26,168	56	8,128,560
Funet	262,144	2,541,568	8,704	288	2,512	0	2,815,216
Mae West	262,144	5,345,280	218,112	512	10,960	24	5,837,032
Oregon	262,144	7,980,032	616,960	1,680	29,256	80	8,890,152
Pacbell	262,144	4,573,184	223,232	272	6,272	24	5,065,128
Paix	262,144	2,754,560	217,600	128	2,376	24	3,236,832
Telstra	262,144	5,849,088	545,280	1,056	16,584	216	6,674,368
Telus	262,144	7,582,720	238,080	1,456	26,184	88	8,110,672
West Attcanada	262,144	7,556,096	291,328	1,472	26,176	56	8,137,272

memory requirements depend on the routing entries distribution and nature, larger routing tables may require smaller memory as shown in Figure 5.9.1, where the routing table for "Funet" requires smaller memory than that for "Aads", although "Funet" has more routing entries than "Aads".

5.9.2 Experiment on Lookup Performance

We have assumed all entries of the routing table are equally likely in our experiments. We have randomly generated (uniform distribution) routing entries using Mersenne Twister [185, 186]. First, we have counted the number of lookups required for each routing entry and tabulated the results in Table 5.4. C_n means that the number of lookup table access is n. For $scheme_1$, the maximum number of lookup table access is 4 and minimum number of lookup table access is 1. Since $Level16$ (Table 5.2) has most entries, 2 lookups are required for most entries, as it is reflected in Table 5.4. Average number of lookups is also $\sim$2 as expected. Since very few entries require

Table 5.3: Memory requirements (%) of the routing tables.

Site	L16	L24	L32	L_CQ16	L_CQ24	L_CQ32
Aads	5.94	88.94	5.02	0.003	0.09	0.001
Att	3.34	96.18	0.13	0.018	0.33	0
East Attcanada	3.22	92.97	3.46	0.018	0.32	0.001
Funet	9.31	90.28	0.31	0.01	0.09	0
Mae West	4.49	91.58	3.74	0.009	0.19	0
Oregon	2.95	89.76	6.94	0.019	0.33	0.001
Pacbell	5.18	90.29	4.41	0.005	0.12	0
Paix	8.1	85.1	6.72	0.004	0.07	0.001
Telstra	3.93	87.64	8.17	0.016	0.25	0.003
Telus	3.23	93.49	2.94	0.018	0.32	0.001
West Attcanada	3.22	92.86	3.58	0.018	0.32	0.001

*Level_CQ*32 (Table 5.2), the number of 4 lookups is also very few.

From Table 5.3, "Funet" routing table requires *Level*16 and *Level*24 most. The "Funet" routing table rarely requires *Level*32, *Level_CQ*16 and *Level_CQ*24, and does not require *Level_CQ*32 at all. Thus the mean number of lookups for the "Funet" routing table is the lowest among all routing table as tabulated in Table 5.4.

Then, We have counted the number of lookups per second as tabulated in Table 5.5. To determine these counts, we have run around 100 times for each routing tables and taken the average.

From Table 5.4, "Funet" routing table should give the best performance. However, since we did our experiments on a general purpose machine with cache memory, due to the caching effect and due to OS intervention, simulation results may not always follow the theoretical results. Although "Funet" routing table does not have the

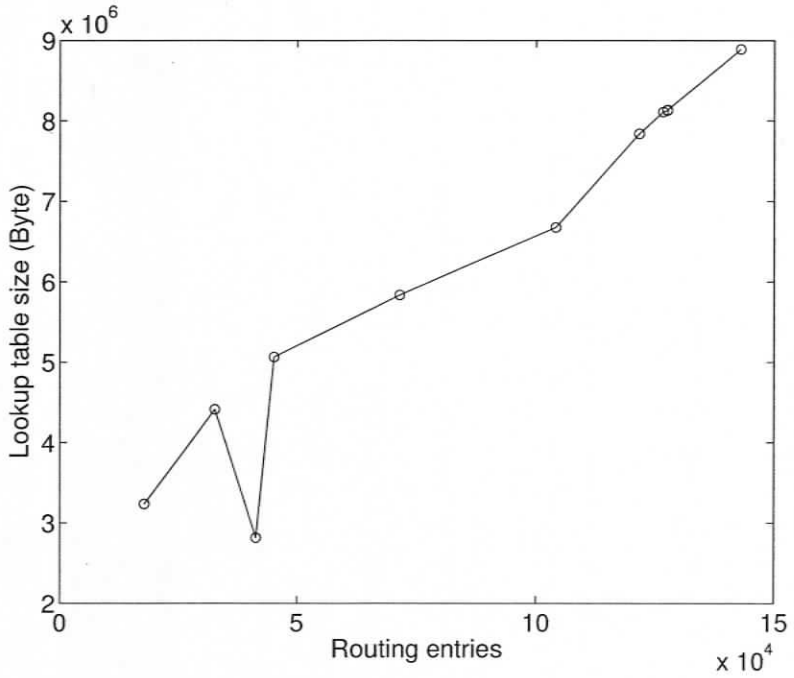


Figure 5.20: Memory requirements vs. Routing entries.

best lookup performance, it is better than most of the routing entries. The maximum lookup performance, we got from our experiments is 8.51 million search/second. The worst case performance, we got is 2.71 million search/second.

5.9.3 Experiment on Update Performance

We have used add to build the lookup table. So, adding all the entries is equal to the build time. We assume 50% deletion and 50% addition in our update time calculation. Our experimental results are tabulated in Table 5.6. To explain the values, we have tabulated the number of memory accesses to delete or add all routing entries in Table 5.7. In this table, we see that "Funet" routing table accesses mostly *Level*16 and *Level*24 than other routing tables. That means "Funet" should have

Table 5.4: Count of lookup table access.

Site	$C_1(\%)$	$C_2(\%)$	$C_3(\%)$	$C_4(\%)$	Mean
Aads	8.92	81.65	9.40	0.027	2
Att	6.78	88.47	4.75	0	1.98
East Attcanada	6.60	87.62	5.78	0	1.99
Funet	16.32	82.24	1.44	0	1.85
Mae West	7.90	85.59	6.49	0.013	1.99
Oregon	5.72	88.52	5.77	0	2
Pacbell	7.96	84.38	7.64	0.02	2
Paix	7.92	78.59	13.44	0.05	2
Telstra	7.73	85.46	6.76	0.05	1.99
Telus	6.677	87.61	5.71	0.01	1.99
West Attcanada	6.53	87.73	5.73	0	1.99

best build and update times as seen in Table 5.6.

5.10 Experiment Using $Scheme_2$

Experiments on the $scheme_2$ would be presented in this section.

5.10.1 Experiment on Memory Requirements

We have tabulated memory requirements of the 11 routing tables in Table 5.8. *Level*16 requires same memory for all routing tables, because *Level*24 has static number of nodes and it is $2^{24} = 16,777,216$. The size of each node is 4 bytes. Thus the total memory requirement for *Level*24 is 67,108,864 bytes. On the other hand, *Level*32 and *Level_CQX* has dynamic number of nodes based on the nature and

Table 5.5: Count per second (Millions per second)

Site	Lookup
Aads	5.66
Att	4.68
East Attcanada	2.71
Funet	5.86
Mae West	8.51
Oregon	4.56
Pacbell	4.14
Paix	6.75
Telstra	5.48
Telus	3.75
West Attcanada	4.19

distribution of the routing entries. However, according to the routing entries' distributions given in Table 5.1, *Level*24 and *Level_CQ*24 should be largest among *LevelX* and *Level_CQX*, respectively, as it is reflected in Table 5.8.

Memory requirements of the routing tables in Table 5.8 are re-tabulated in Table 5.9 as a percentage of the total memory requirement.

Generally, larger routing tables require larger memory. Since in our technique, memory requirements depend on the routing entries distribution and nature, larger routing tables may require smaller memory as shown in Figure 5.10.1, where the routing table for "Funet" requires smaller memory than that for "Aads", although "Funet" has more routing entries than "Aads".

Table 5.6: Count per second (Millions per second)

Site	Delete	Add/Build	Update
Aads	4.44	1.18	1.87
Att	6.02	1.47	2.37
East Attcanada	4.51	1.46	2.21
Funet	4.90	2.70	3.48
Mae West	7.46	1.87	2.99
Oregon	4.85	1.60	2.41
Pacbell	5.46	1.43	2.27
Paix	4.32	1.07	1.72
Telstra	6.88	2.07	3.18
Telus	5.96	1.78	2.74
West Attcanada	3.22	1.54	2.09

5.10.2 Experiment on Lookup Performance

We have assumed all entries are equally likely in our experiments. We have randomly generated (uniform distribution) routing entries using Mersenne Twister [185, 186]. First, we have counted the number of lookups required for each routing entries and tabulated the results in Table 5.10. For this scheme, the maximum number of lookup table access is 3 and minimum number of lookup table access is 1. Since $Level24$ (Table 5.8) has most entries, 1 lookups are required for most entries, as it is reflected in Table 5.10. Average number of lookups is also $\sim$1 as expected. Since very few entries require $Level\_CQ32$ (Table 5.8), the number of 3 lookups is also very few.

From Table 5.9, "Funet" routing table requires $Level24$ most. The "Funet" routing table rarely requires $Level32$ and $Level\_CQ24$, and does not require $Level\_CQ32$ at all. Thus the mean number of lookups for the "Funet" routing table is the lowest

Table 5.7: Count of memory accesses for building the routing table

Site	Total	L16	L24	L32	L_Q16	L_Q24	L_Q32
Aads	65,183	49.87	45.4	3.08	0.02	1.62	0.01
Att	240,954	50.51	47	0.01	0.09	2.39	0
East Attcanada	254,070	50.21	46.81	0.62	0.08	2.28	0
Funet	76,509	54.02	45.18	0.03	0.02	0.75	0
Mae West	141,651	50.35	46.32	1.35	0.05	1.93	0.01
Oregon	285,836	49.99	47.06	0.79	0.07	2.09	0
Pacbell	90,241	50.07	46.04	2.1	0.04	1.74	0.01
Paix	36,530	48.63	44.76	4.94	0.02	1.62	0.02
Telstra	207,280	50.22	46.28	1.52	0.05	1.9	0.02
Telus	252,161	50.24	46.81	0.51	0.08	2.36	0
West Attcanada	254,131	50.2	46.85	0.6	0.07	2.28	0

among all routing table as tabulated in Table 5.10.

Then, We have counted the number of lookups per second as tabulated in Table 5.11. To determine these counts, we have run around 100 times for each routing tables and taken the average. From Table 5.10, "Funet" routing table should give the best performance as reflected in Table 5.11. The maximum lookup performance, we got from our experiments is 8.51 million search/second. The worst case performance, we got is 2.71 million search/second.

5.10.3 Experiment on Update Performance

We have used add to build the lookup table. So, adding all the entries is equal to the build time. We assume 50% deletion and 50% addition in our update time calculation. Our experimental results are tabulated in Table 5.12. To explain the

Table 5.8: Memory requirements (Bytes) of the routing tables.

Site	Level24	Level32	Level24_Q	Level32_Q	Total
Aads	67,108,864	221,696	10,272	24	67,340,856
Att	67,108,864	9,728	62,000	0	67,180,592
East Attcanada	67,108,864	233,472	63,184	56	67,405,576
Funet	67,108,864	7,680	6,624	0	67,123,168
Mae West	67,108,864	218,112	26,800	24	67,353,800
Oregon	67,108,864	539,136	70,944	80	67,719,024
Pacbell	67,108,864	223,232	15,216	24	67,347,336
Paix	67,108,864	217,600	6,048	24	67,332,536
Telstra	67,108,864	486,400	40,832	216	67,636,312
Telus	67,108,864	228,352	63,600	88	67,400,904
West Attcanada	67,108,864	240,128	63,200	56	67,412,248

values, we have tabulated the number of memory accesses to delete or add all routing entries in Table 5.13. In this table, we see that "Funet" routing table accesses mostly *Level*24 than other routing tables. That means "Funet" should have best build and update times as seen in Table 5.12.

5.11 Time Complexity Analysis

Assume,

- p_{16} is the probability of being *Level*16.
- p_{CQ16} is the probability of being *Level_CQ*16.
- p_{24} is the probability of being *Level*24.

Table 5.9: Memory requirements (%) of the routing tables.

Site	Level24	Level32	Level24_Q	Level32_Q
Aads	99.66	0.33	0.02	0
Att	99.89	0.01	0.09	0
East Attcanada	99.56	0.35	0.09	0.0001
Funet	99.98	0.01	0.01	0
Mae West	99.64	0.32	0.04	0
Oregon	99.1	0.8	0.1	0.0001
Pacbell	99.65	0.33	0.02	0
Paix	99.67	0.32	0.01	0
Telstra	99.22	0.72	0.06	0.0003
Telus	99.57	0.34	0.09	0.0001
West Attcanada	99.55	0.36	0.09	0.0001

- p_{CQ24} is the probability of being $Level\_CQ24$.
- p_{32} is the probability of being $Level32$.
- p_{CQ32} is the probability of being $Level\_CQ32$.

Then time complexity for table lookup for 3-stride trie ($scheme_1$) is given by,

$$T_{lookup} = 1 \times a + 2 \times (b + c) + 3 \times (d + e) + 4 \times f \tag{5.1}$$

Then time complexity for addition to the table for 3-stride trie ($scheme_1$) is given by,

$$T_{add} = 1 \times a + 2 \times (b^{cover} \times 2^4 + c) + 3 \times (d^{cover} \times 2^3 + e) + 4 \times f^{cover} \times 2^3 \tag{5.2}$$

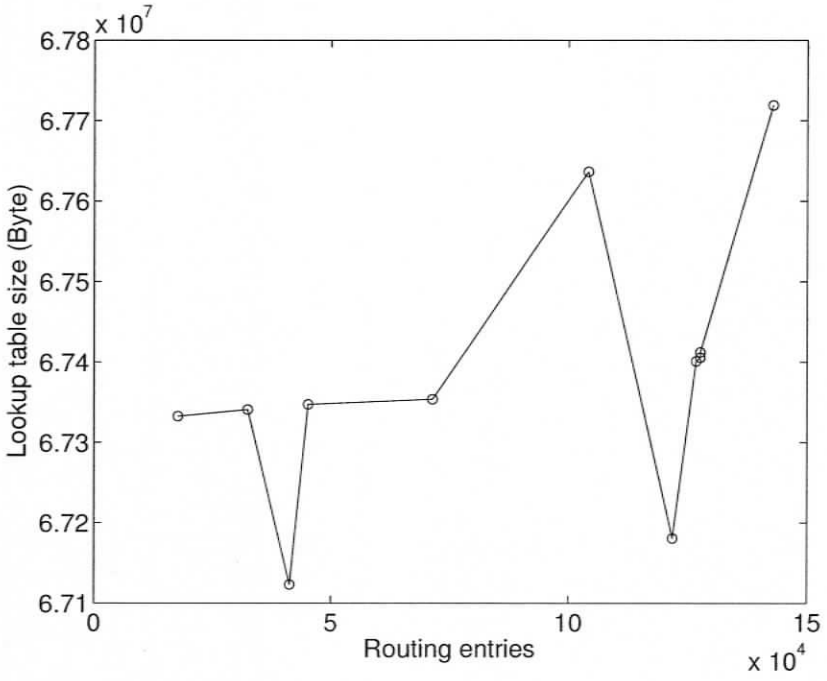


Figure 5.21: Memory requirements vs. Routing entries.

Then time complexity to remove from the table for 3-stride trie ($scheme_1$) is given by,

$$T_{remove} = 1 \times a + 2 \times (b^{cover} + c) + 3 \times (d^{cover} + e) + 4 \times f^{cover} \qquad (5.3)$$

If we look at the simulation results in Section 5.9 and Section 5.10, we will see $T_{lookup} < T_{remove} < T_{add}$ most of the time. There may have few exceptions because of the distribution and nature of routing table entries.

5.12 Conclusion

In this chapter, we have proposed a novel IP-lookup algorithm that is extremely fast in IP-lookup and table update. This chapter first explicitly elaborates the solution

Table 5.10: Count of lookup table access.

Site	$C_1(\%)$	$C_2(\%)$	$C_3(\%)$	Mean
Aads	89.68	10.28	0.03	1.10
Att	94.46	5.54	0	1.06
East Attcanada	90.47	9.52	0.01	1.10
Funet	98.12	1.88	0	1.02
Mae West	92.59	7.40	0.01	1.07
Oregon	90.53	9.46	0.01	1.09
Pacbell	91.50	8.49	0.01	1.09
Paix	85.23	14.73	0.04	1.15
Telstra	92.11	7.85	0.04	1.08
Telus	91.76	8.22	0.02	1.08
West Attcanada	90.27	9.72	0.01	1.10

of a problem in expanding IP addresses. We give mathematical expression to determine the performance theoretically. We have done extensive numerical simulations to determine the performance of our proposed algorithm and also to determine best values for different parameters for the IP-lookup. The simulation results show that our proposed technique performs better than existing techniques in terms of lookup and update times and number of total memory access during IP-lookup. Few techniques have better time performance, although the number of memory access of their techniques are larger than that of ours. The reason of this is that they used computers with better configurations. Even though, their update times are much higher than that of ours. However, our proposed technique requires larger memory than others. But the memory requirement is quite acceptable considering the current memory price.

Table 5.11: Count per second (Millions per second).

Site	Lookup
Aads	4.31
Att	3.51
East Attcanada	5.45
Funet	9.78
Mae West	6.87
Oregon	4.91
Pacbell	6.29
Paix	5.30
Telstra	6.61
Telus	2.83
West Attcanada	4.01

We have addressed the cases where same IP-address with different PL. Our experimental results shows that $scheme_1$, maximum and minimum lookups are 8.51 and 2.71 million per second, respectively. Memory requirements are 2.60MB (min) and 8.47MB (max). Update count are 1.72 million per second (min) and 3.48 million per second (max). For scheme 2, maximum and minimum lookups are 9.78 and 2.83 million per second, respectively. Memory requirements are 64.01MB (min) and 64.58MB (max). Update count are 0.50 million per second (min) and 1.02 million per second (max).

Comparing between $scheme_1$ and $scheme_2$, $scheme_1$ has better update time, memory requirement, and has comparable or slightly less lookup performance than those of $scheme_2$. The lookup table access for $scheme_2$ is half of the $scheme_1$. But the lookup performance of $scheme_2$ is not double of $scheme_1$. This occurs for two

Table 5.12: Count per second (Millions per second).

Site	Delete	Add/Build	Update
Aads	2.15	0.47	0.77
Att	2.95	0.53	0.91
East Attcanada	2.86	0.54	0.90
Funet	4.35	0.31	0.58
Mae West	4.51	0.58	1.02
Oregon	2.25	0.57	0.91
Pacbell	2.95	0.56	0.94
Paix	2.36	0.52	0.85
Telstra	2.82	0.53	0.89
Telus	1.95	0.28	0.50
West Attcanada	2.25	0.52	0.85

reasons:

- Although the lookup table access in $scheme_1$ is double of that in $scheme_2$, other memory access during the lookup process is not double for $scheme_1$.
- 16-bit ($scheme_1$) memory access is faster that 24-bit ($scheme_2$) memory access in a 32-bit machine, used in the simulation.

The lookup performance of $scheme_2$ is only slightly higher than that of $scheme_1$ at the expense of memory requirement. If we make 1-level scheme ($scheme_3$), the memory requirement would be 16.78 TB, making it infeasible for practical implementation. Considering, lookup, memory and update performances, $scheme_1$ is the best choice.

Our technique can easily be expanded for IPV6 [187] without changing any concepts outlined in this chapter.

Table 5.13: Count of memory accesses for building the routing table.

Site	Level24	Level32	Level24_Q	Level32_Q
Aads	90.62	5.51	3.84	0.03
Att	89.29	0.06	10.65	0
East Attcanada	83.6	6.94	9.44	0.02
Funet	98.15	0.06	1.79	0
Mae West	52.2	17.3	30.43	0.06
Oregon	46.58	24.79	28.57	0.06
Pacbell	92.15	3.77	4.06	0.01
Paix	87.1	9.05	3.81	0.03
Telstra	82.38	6.76	10.78	0.08
Telus	85.37	4.52	10.07	0.04
West Attcanada	83.31	7.22	9.44	0.02

Chapter 6

Best Matching

In this chapter, we propose a novel Best Match technique required to detect best-matched English words of obfuscated spam words. This chapter is organized as follows. In Section 6.1, we mention few existing techniques for Best Match. In Section 6.2, we formulate a system that can clean-up obfuscated words created by the five techniques mentioned in Table 1.2. In Section 6.3, we present an NFA to build the dictionary. In Section 6.4, we describe the preprocessing required for the obfuscated spam words. In Section 6.5, we discuss a technique required to detect the best-matched word in the NFA. In Section 6.7, we do extensive numerical simulations to see the effects of different design parameters on detection accuracy and also the time complexity of the system based on our proposed technique. In Section 6.8, we present the time complexity analysis that comply the results in Section 6.7. In Section 6.9, we conclude our work with future research direction.

6.1 Related Works

There are thousand of works for correcting unintentional (e.g. misspelled words) obfuscated words [80, 188]. Few techniques in this area are mentioned below.

Different dictionary partitioning techniques are described for best matching in [189]. However their best techniques cannot deal the word that has more than one error or whose first character is obfuscated. Similar problems exist in the work described in [190] where S. Didanova used linear array for spelling correction. To speed up the operation, she presented parallel architecture in her paper.

The ternary search tree is used in [191] to store the dictionary. This technique used some heuristic approaches on the ternary search tree for best match intended for search engine queries. However, their work is not versatile, i.e. only deals with few specific types of errors.

Hidden Markov Model (HMM) is used in [79] to correct obfuscated spam words. However, HMM cannot correct all word boundary errors. For example, "A27_off" would not be decoded correctly. In fact, if the first character is missing or substituted, HMM seems to produce erroneous results. It is also not clear whether "actr" would be decoded as "act" or "actor". It is also not described how non-alphabetic characters would be treated in the HMM. There is a similar work in [192] where Ristad et al. proposed a memoryless stochastic model for string-edit distance calculation.

Finite automata techniques in [193–196] can be used for Best Matching. But the time requirement would be very high for a large dictionary. Universal deterministic Levenshtein automaton is used in [197] to determine the Levenshtein distance between two words: P and T. The input of the automaton is a sequence of bit-vectors computed from P and T. Since The dictionary word, T is stored in another finite-sate automaton, two automata have to be traversed to get the matching result. Also in their technique, different Levenshtein automata are required for different threshold

values.

The main problem of the above techniques is that they did not consider all the techniques used by spammers as mentioned in Section 1.4.5.

6.2 The Proposed Technique

To explain our proposed technique, we divide all characters into three categories:

1. alphabetic characters, Γ,
2. characters that look alike alphabetic characters, Ψ, and
3. non-alphabetic characters that do not look alike alphabetic characters, Φ.

If Λ denotes all types of characters, then

$$\Lambda = \Gamma + \Psi + \Phi \tag{6.1}$$

We use an NFA [198] based system that works faster than the method described in [195, 196] and can deal with all the obfuscating techniques mentioned in Section 1.4.5.

Dynamic programming is used to navigate the dictionary in finite automaton [193–196]. To reduce the exploration space of the dynamic programming, pruning is used by some researchers [193, 194].

Our proposed technique has three phases:

- Building the dictionary.
- Preprocess obfuscated words: this phase works on Ψ and Φ.
- Detect best-matched words: the detection phase works on Γ.

The following sections discuss these steps.

6.3 Building English Dictionary

We have used NFA to build the English dictionary. We draw in Figure 6.1 an NFA that holds the English dictionary which is used for detecting best-matched word.

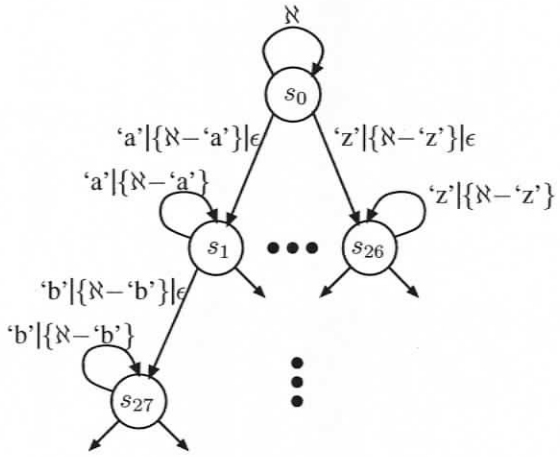


Figure 6.1: Generalized automaton to build the English dictionary.

To build the automaton, we consider only the transitions without cost. The meaning of cost will be discussed in Section 6.5. At this moment, we assume that Γ and ϵ transitions are associated with costs. Thus we consider here only transitions for alphabetic characters based on English dictionary. Costly transitions will be dealt with during the detection of the best-matched words in Section 6.5. We implement each state as a structure of three fields: $\mathcal{P}$, S, and F. $\mathcal{P}$ is an array of pointers for next states. The size of the array equals the number of English characters, i.e., 26. S holds the resulting best matched string word. F is 2 bit flag whose significance is written in Table 6.1.

For example, Figure 6.2 shows the automaton of three English words: "off", "offer", and "few".

Table 6.1: Significance of F of the NFA state.

F	Meaning
$F = 0$	$\mathcal{P}$ contains valid next state pointers.
$F = 1$	S contains valid output string.
$F = 2$	$\mathcal{P}$ contains valid next state pointers and S contains valid output string. This situation occurs for the cases like "off" and "offer".
$F = 3$	unused

6.4 Preprocessing Obfuscated Words

We preprocess obfuscated spam words to take off the non-alphabetic characters that are not defined in the rule-set. Preprocessing works on some user-defined rules. The user decides whether a non-alphabetic character belongs to Ψ or Φ. For example, for some cases, | belongs to Φ; while for some cases | belongs to Ψ since it looks like i or l. Even an alphabetic character may look similar to another alphabetic character. For example, I and l may be substituted by one another without much confusion. Even some users may think that g and q may be substituted without much confusion as well. All these look-alike characters fall in Ψ and must be defined by the user for preprocessing.

Preprocessing (i) deletes all the characters in Φ from the obfuscated word, and (ii) substitutes all the characters in Ψ by the ones defined by the user. Assume the following rules are defined by the user: $\{(,)\} \in \Phi$, $\{|\} \in \Psi$, and $\{| \rightarrow \text{i, l}\}$, where $\longrightarrow$ means "substituted by". If the obfuscated input is "(l)(|)(v)(e)", then preprocessing produces "l{i, l}ve", i.e., the word may be "llve" or "live". The matching phase will decide the best-matched word.

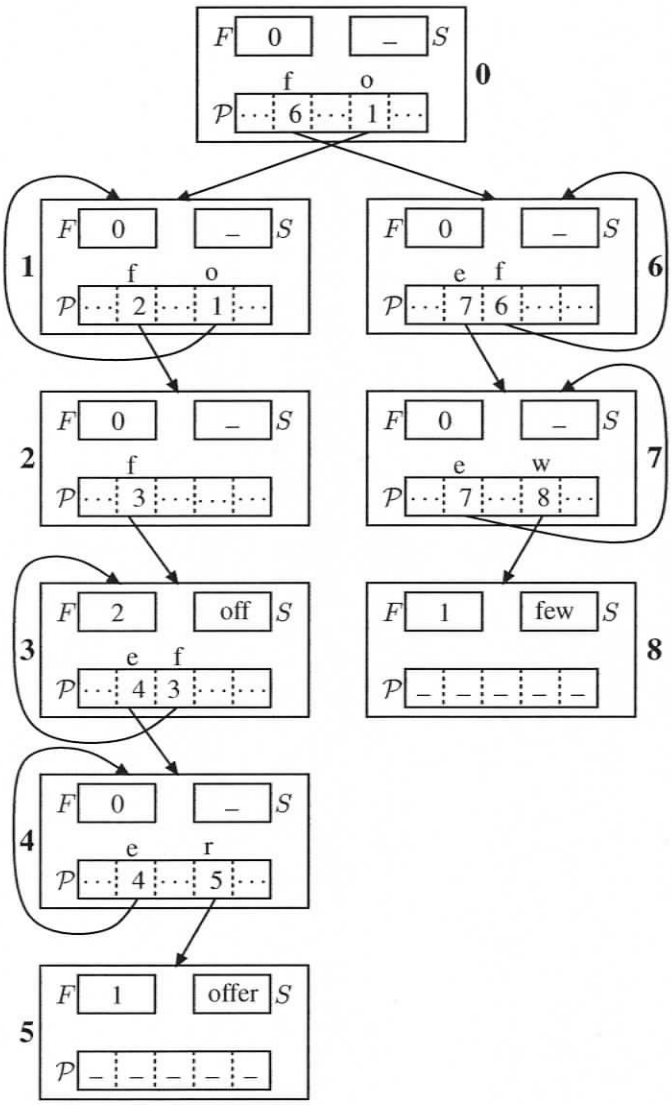


Figure 6.2: Automaton built for "off", "offer", and "few".

6.4.1 Implementation

To understand the following discussion, we have to know the types of substitutions. There are two types of substitutions:

Simple:

- One-to-one: '0' $\longrightarrow$ 'o'.
- One-to-many: '|' $\longrightarrow$ 'i' or 'l'.
- Many-to-one: '><' $\longrightarrow$ 'x'.
- Many-to-many: '\/\/' $\longrightarrow$ 'v' or 'w'.

Compound: Sets of substitution rules, where the first character of the substituted strings in the whole set is same. For example, assume '!' is the first character of the substituted strings in a set. Then that set may contain the following rules:

'!' $\longrightarrow$ 'i' or 'l'
'!<' $\longrightarrow$ 'k'.
'!\/!' $\longrightarrow$ 'm'.
...

For the preprocessing phase, we use two structures: T_1 and T_2. T_1 is a hash table of 8-bit ASCII characters [184]. During preprocessing, T_1 is indexed by the ASCII value of a character of the obfuscated spam word being preprocessed. T_1 has three fields: F, S, and A. F is a flag of 2 bits whose significance is written in Table 6.2. T_2, used for many-to-one, many-to-many, and compound substitutions, has three fields: S_1, S_2, and A. S_1 holds other than first characters for many-to-one, many-to-many, and compound substitutions. S_2 holds the string to be substituted for S_1. A holds the next location in T_2 for compound substitutions.

For example, assume the rule set is as given in Table 6.3: T_1 and T_2 look like Figure 6.3 and Figure 6.4 for the rule set given in Table 6.3.

To prepossess the obfuscated spam word, we use a simple algorithm given in [86].

Table 6.2: Significance of F of the NFA state.

F	Meaning
$F = 0$	delete the character; the character is not a legitimate character.
$F = 1$	return string S; the character is legitimate or to be substituted by S.
$F = 2$	go to the location A of T_2; the character is the first character of many-to-one, many-to-many, and compound substitutions.
$F = 3$	unused

Table 6.3: Rule set

Input characters	substituted to
0	o
|	{i, l}
><	x
\/\/	{v, w}
!<	k
!	{i, l}

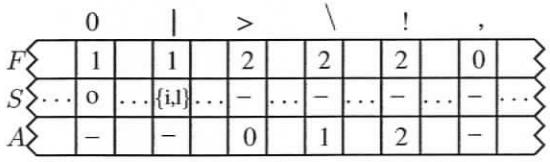


Figure 6.3: Structure T_1.

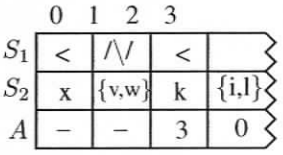


Figure 6.4: Structure T_2.

6.5 Detecting Best-Matched Words

We have used dynamic programming with state pruning to navigate the NFA to detect the best-matched words. Dynamic programming is one of the techniques to navigate the NFA [196, 199]. But the main problem of dynamic programming is its large state space [200, 201]. This large state space increases computational time and computer storage requirements. One of the methods to remove this problem is to prune some of the states based on certain criteria:

- Prune the state, if the cost is greater than threshold, Θ.
- Prune the state for "low cost", if the cost is greater than the cost for the already found word.

To prune the state during dynamic programming, we have to accumulate the cost as the simulation is running from s_0.

We have used Levenshtein distance [202] to determine the cost during the NFA navigation. The Levenshtein distance between two strings, T and P is the minimal

number of edit operations (insert, delete, and replace) required to convert P into T, assuming P and T are obfuscated spam word and legitimate word, respectively. The automaton in Figure 6.1 is built using English words including T. Assume we are in the initial state s_0 of the automaton in Figure 6.1 in our way to find the best matched word for P. Now four different situations may occur:

- If p_0 (first character of P) is 'a', then the next state will be s_1. The cost for this transition is 0 according to Levenshtein distance. The character index of P will be increased by 1 for this transition, i.e., p_1 will be checked next.

- If we assume that the first character is **deleted** to obfuscate P, then the next states will be s_1, s_2, $\cdots$, s_{26}. This corresponds to the ϵ-transitions from s_0 in Figure 6.1. The cost for this transition is 1 according to Levenshtein distance. The character index of P will remain in its current value, i.e., p_0 will be checked again.

- If we assume that the first character is **inserted** to obfuscate P, then the next state will remain at s_0. This corresponds to the Γ-transitions from s_0 to s_0 in Figure 6.1. The cost for this transition is 1 according to Levenshtein distance. The character index of P will be increased by 1 for this transition, i.e., p_1 will be checked next.

- If we assume that p_0 is **substituted** by a to obfuscate P, then the next states will be s_2, $\cdots$, s_{26}. This corresponds to the $\Gamma-$ 'b', $\Gamma-$ 'c', $\cdots$ transitions from s_0 in Figure 6.1. The cost for this transition is 1 according to Levenshtein distance. The character index of P will be increased by 1 for this transition, i.e., p_1 will be checked next.

As we continue our search for the best matched word for P in the automaton (Figure 6.1), we have to deal the above mentioned four situations in every state of the

automaton.

Above discussion does not include anything on how to deal with word boundary problems. We simulate the automaton for $P = p_0p_1 \cdots p_m$, $p_1p_2 \cdots p_m$, $p_2p_3 \cdots p_m$, $\cdots$ and collects the best matched words $\mathcal{W}_0$, $\mathcal{W}_1$, $\mathcal{W}_2$, $\cdots$ for costs $\mathcal{C}_0$, $\mathcal{C}_1$, $\mathcal{C}_2$, $\cdots$, respectively. For "low cost", the solution is that word which corresponds to the lowest cost. If the simulation produces two or more best matched words for the lowest cost, then the solution is the longest word among the corresponding best matched words. For "longer match", the solution is the longest word among the best matched words. If two or more best matched words have the same longest length, then the solution is the lowest cost word.

6.5.1 Implementation

Assume $\mathcal{S}$ is the next state from s_0 for the input p_0. SL is the list of all possible next states from s_0. Then assuming that p_0 is the correct input, the best matched word is,

$$\mathcal{W}_0 \leftarrow DP(\mathcal{S}, \mathcal{C}_0, W\_index + offset)$$

where $\mathcal{C}$ ($\mathcal{C}_0 = \mathcal{C}$) is the Levenshtein distance so far, W_index is the current index in the preprocessed word, and $W\_index + offset$ is next index in the preprocessed word to be processed. Generally the value of $offset$ is 1, however for the cases like {i, l}, $offset = 5$.

If we assume that the first character is deleted, then the best matched word is,

$$\mathcal{W}_1 \leftarrow DP(SL, \mathcal{C}_1, W\_index)$$

where $\mathcal{C}_1 = \mathcal{C} + 1$.

If we assume that the first character is inserted, then the best matched word is,

$$\mathcal{W}_2 \leftarrow DP(s_0, \mathcal{C}_2, W\_index + offset)$$

where $\mathcal{C}_2 = \mathcal{C} + 1$.

If we assume that p_0 is substituted, then the best matched word is,

$$\mathcal{W}_3 \leftarrow DP(\{SL - \mathcal{S}\}, \mathcal{C}_3, W\_index + offset)$$

where $\mathcal{C}_3 = \mathcal{C} + 1$.

Now, if we are looking for low cost match, then

$$\mathcal{W} \leftarrow Min(\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3)$$

If we are looking for longer match, then

$$\mathcal{W} \leftarrow Max(\mathcal{W}_0, \mathcal{W}_1, \mathcal{W}_2, \mathcal{W}_3)$$

The above statements are recursively used until we get the best-matched word. In general, we have to replace s_0 by the current state.

Now to prune certain states, we have to check $\mathcal{C}$ and return from DP by

$$\textbf{if } \mathcal{C} > \Theta \textbf{ then return}$$

before proceeding with calculating $\mathcal{W}$s. If we are looking for the low cost, then we can further prune states by,

- calculation of $\mathcal{W}_1$ should be done if

$$\mathcal{C}_0 > \mathcal{C}$$

- calculation of $\mathcal{W}_2$ should be done if

$$\mathcal{C}_0 > \mathcal{C} \textbf{ AND } \mathcal{C}_1 > \mathcal{C} + 1$$

- calculation of $\mathcal{W}_3$ should be done if

$$\mathcal{C}_0 > \mathcal{C} \textbf{ AND } \mathcal{C}_1 > \mathcal{C} + 1 \textbf{ AND } \mathcal{C}_2 > \mathcal{C} + 1$$

To deal with word boundary, we have to call DP with different values of W_index, where W_index varies from 0 to $m - 1$ (m = length of the preprocessed word). The pseudo-code of dynamic programming algorithm is given in [86].

6.6 Example

To give an example, how the detection process described in Section 6.5 works on the automaton in Figure 6.1, we assume that our dictionary consists of three words: "off", "offer", and "few". The automaton of these three words is drawn in Figure 6.2. Also assume that the obfuscated word is "ofer". The running space of the dynamic programming described in Section 6.5.1 for "ofer" is drawn in Figure 6.5. To differentiate between two f, we denote the second f as $\int$ in Figure 6.5. $\times$ means that the state is pruned as the cost becomes greater than Θ. $\otimes$ means the dead state that can not be reached because of the character to be considered.

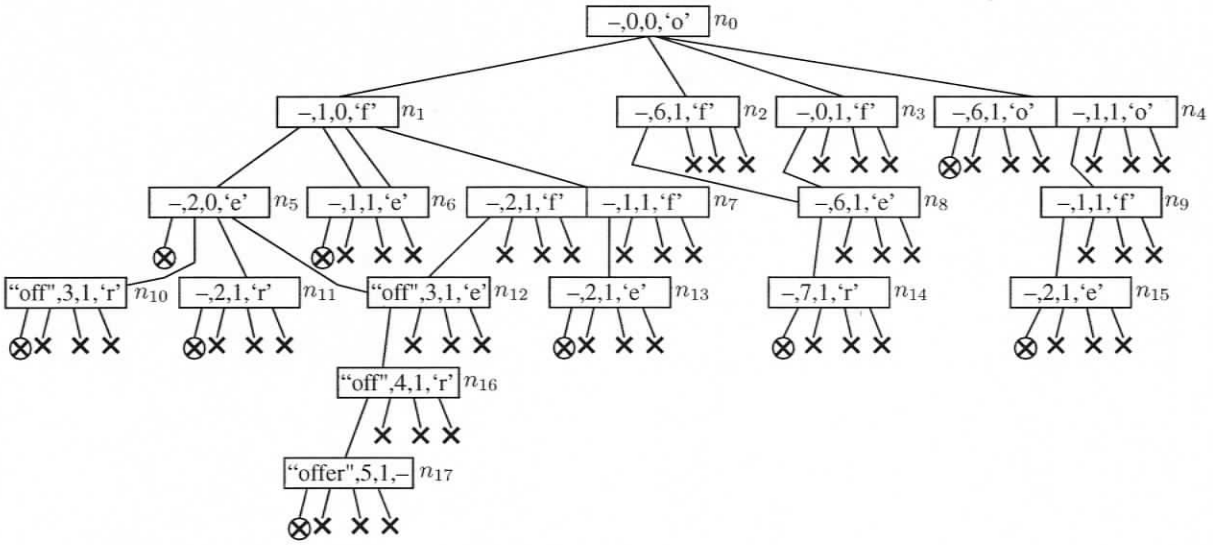


Figure 6.5: Running space of the dynamic programming for the obfuscated word 'ofer'.

In Figure 6.5, there are four outgoing paths from each state. These are for normal, deletion, insertion, and substitution, that means, the matching algorithm is recursively called for four times. The order of these calls has an impact on the time complexity of the matching algorithm as will be discussed in Section 6.7. For our

example here, we follow the order: normal, substitution, deletion, and insertion. In Figure 6.5, leftmost outgoing path is for normal, second leftmost is for substitution, and so on. Θ of our example is 1. Each node in the running space has the tuple of the following four values: $<\mathcal{W}$, corresponding state in the automaton, cost (Levenshtein distance) from n_0 to the current state, and current character to be processed>.

The simulation starts with the values: <–, 0, 0, 'o'>. The simulation continues with the first recursive call for the *normal* case that follows the leftmost path from n_0 to n_1 in Figure 6.5. The value becomes <–, 1, 0, 'f'>. The simulation goes to n_5 (<–, 1, 0, 'f'>) from n_1 for the *normal* case. The simulation goes to a dead end as it cannot continue from n_5 for the *normal* case. This is because the case that there is no entry in $\mathcal{P}$ of n_2 of the automaton in Figure 6.2. If we assume that the next character is substituted by 'e', then the next state would be n_{10}. $\mathcal{W}$ in n_{10} is "off" with cost equals to 1 and 1 character to be checked. Thus for this $\mathcal{W}$, the Levenshtein distance is 2 which is greater than Θ (1). The simulation comes to a dead end for the *normal* operation. Since the cost already equals to Θ, the states for *substitution*, *insertion*, and *deletion* operations are pruned, as they the costs in those states are greater than Θ. So, the simulation backtracks to n_5 to check the states for *insertion* and *deletion* operations. The *insertion* operation leads the simulation to n_{11} without any conclusion. The *deletion* operation leads the simulation to n_{12}. Since the cost of n_{12} is already 1, the states *substitution*, *insertion*, and *deletion* operations are pruned. From n_{12}, the simulation follows the path for *normal* operation to n_{17} through n_{16}. In n_{17}, $\mathcal{W}$ is "offer" with cost equals to Θ. Thus, "offer" is one of the possible matched words for the obfuscated word "ofer" (for $\Theta = 1$). Therefore, all four paths from n_5 are tried. The simulation backtracks to n_1 to try the states for *substitution*, *insertion*, and *deletion* operations. Both *substitution* and *insertion* operations lead the simulation to n_6 without any conclusion. But *deletion* operation leads the simulation to n_7 with the values: <–, 2,1, 1, 'f'>. <–, 1, 1, 'f'> of n_7 leads the simulation to n_{13} without

any conclusion. <–, 2, 1, 'f'> of n_7 leads the simulation to the same path as n_5 does and concludes that the matched word is "offer" for the obfuscated word "ofer" (for $\Theta = 1$). Same descriptions can be applied for the *substitution*, *insertion*, and *deletion* operations from n_0.

6.7 Experiment

To experiment our proposed technique, we have used three different datasets: (i) manually 100 obfuscations of "viagra" (dataset_1), (ii) manually 100 obfuscations of 20 most obfuscated spam words (5 per word) [203] (dataset_2), and (iii) around 400 unique obfuscated spam words obtained from around 20,000 e-mails (dataset_3). We have collected these e-mails from the Ling-Spam corpus available in [204] and e-mails we got in our e-mail accounts. First two datasets are mainly used to determine accuracy of our system and the last one is mainly used to determine performance evaluation of our system.

6.7.1 Size of the Automaton

We determine the sizes of the automaton (Section 6.3) for seven different English word-sets found in [205] and tabulate our observations in Table 6.4. The relationship between number of words in word-sets and the required number of states in the automaton is not always linear, because of the character distribution in words of the word-sets. For example, word-set_7 has more words than word-set_6, but automaton states of word-set_7 are less than that of word-set_6.

The size of each state is 7 bytes in our implementation. Then the memory requirement for the automaton for word-set_6 (highest value among seven word-sets in Table 6.4) is 909 KB which is reasonable considering current price and availability

Table 6.4: Size of the automaton for English dictionary.

Word-set	number of Words	Required number of states	Required memory (KB)
1	743	3,171	22
2	1,899	5,689	39
3	6,688	23,178	158
4	19,708	79,831	546
5	20,864	67,320	460
6	37,616	132,959	909
7	47,158	98,752	675

of the memory.

6.7.2 Effects of design parameters on detection accuracy

To determine the effects of design parameters on detection accuracy, we have used all three datasets mentioned above. The considered design parameters are tabulated in Table 6.5 along with the symbols used to represent those parameters. We have not increased the value of Θ beyond 3, because if the Levenshtein distance between an obfuscated word and the original word is greater than 2, the reader would most probably be confused with the obfuscated word. This will oppose the principle of spams. For this reason, the spammers does not like to obfuscate more than twice in a word. The experiments with the real spams also show that $\Theta = 2$ gives the best detection accuracy (Figs. 6.14, 6.15, 6.16, and 6.17).

For each simulation, we have collected three types of data: (i) detection accuracy, (ii) false detection, and (iii) no detection (fail to make any decision). We will see in the following how the parameters in Table 6.5 affects these three types of data.

Table 6.5: Design parameters considered in our experiments.

Design parameters	Symbols
Longer match	L
Low cost	L'
Word boundary (active)	W
Word boundary (disabled)	W'
Rule set (used)	R
Rule set (unused)	R'
Threshold	$\Theta(0 \leq \Theta \leq 3)$

Experiment on Dataset$_1$

Figure 6.6 shows the experimental results on dataset$_1$ for L'. Using $\Theta > 2$ does not increase the detection accuracy at all. Using rule set gives the best detection accuracy as it lowers the false detection. However, word boundary does the opposite as it increases the false detection.

Figure 6.7 shows the experimental results on dataset$_1$ for L. Using $\Theta > 2$ increases the detection accuracy when the rule set is not used. To understand this, assume "vi(@)gr@" as the obfuscated word. Without the rule set, the preprocessed word becomes "vigr". For low cost, our system produces "big" with cost 1 ('b' is substituted by 'v'). For longer match, it produces "viagra" with cost 2 (deletion of 2 a's) which is correct best-matched word for "vi(@)gr@". Like Figure 6.6, word boundary increases the false detection.

Figure 6.8 and Figure 6.9 show the experimental results on dataset$_1$ for R' and R, respectively. These are to see the effects of L on the system performance. Both figures prove that longer match gives better results.

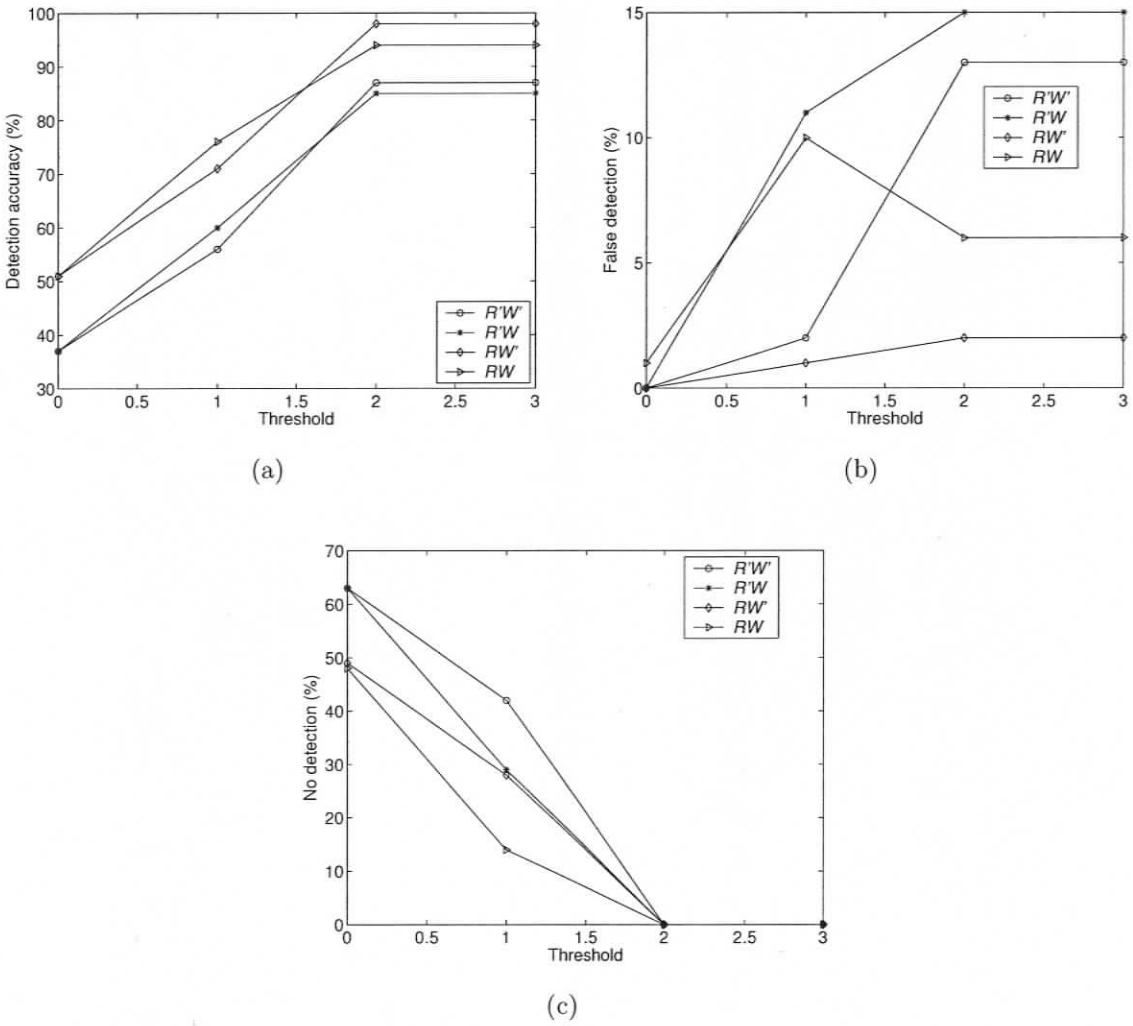


Figure 6.6: Experimental results on dataset$_1$ for L': (a) detection accuracy, (b) false detection, (c) no detection.

Experiment on Dataset$_2$

Figure 6.10 shows the experimental results on dataset$_2$ for L'. Using $\Theta > 2$ does not increase the detection accuracy significantly, on the other hand, it decreases the detection accuracy for some cases. Like experiments with dataset$_1$, rule set gives the

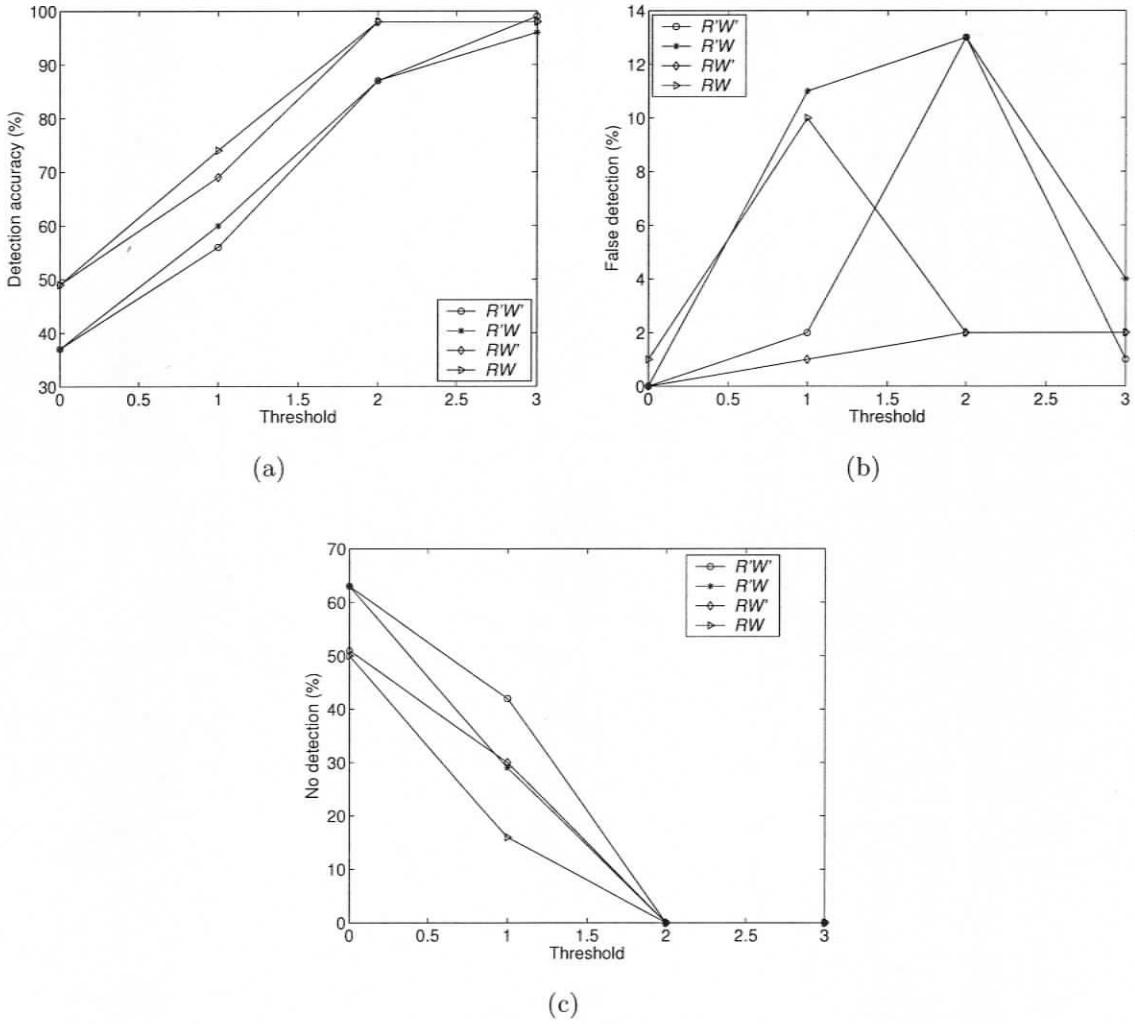


Figure 6.7: Experimental results on dataset$_1$ for L: (a) detection accuracy, (b) false detection, (c) no detection.

best detection accuracy as it lowers the false detection. Similarly, word boundary does the opposite as it increases the false detection.

Figure 6.11 shows the experimental results on dataset$_2$ for L. Using $\Theta > 2$ increases the detection accuracy most of the time, but decreases the rate for RW' (see

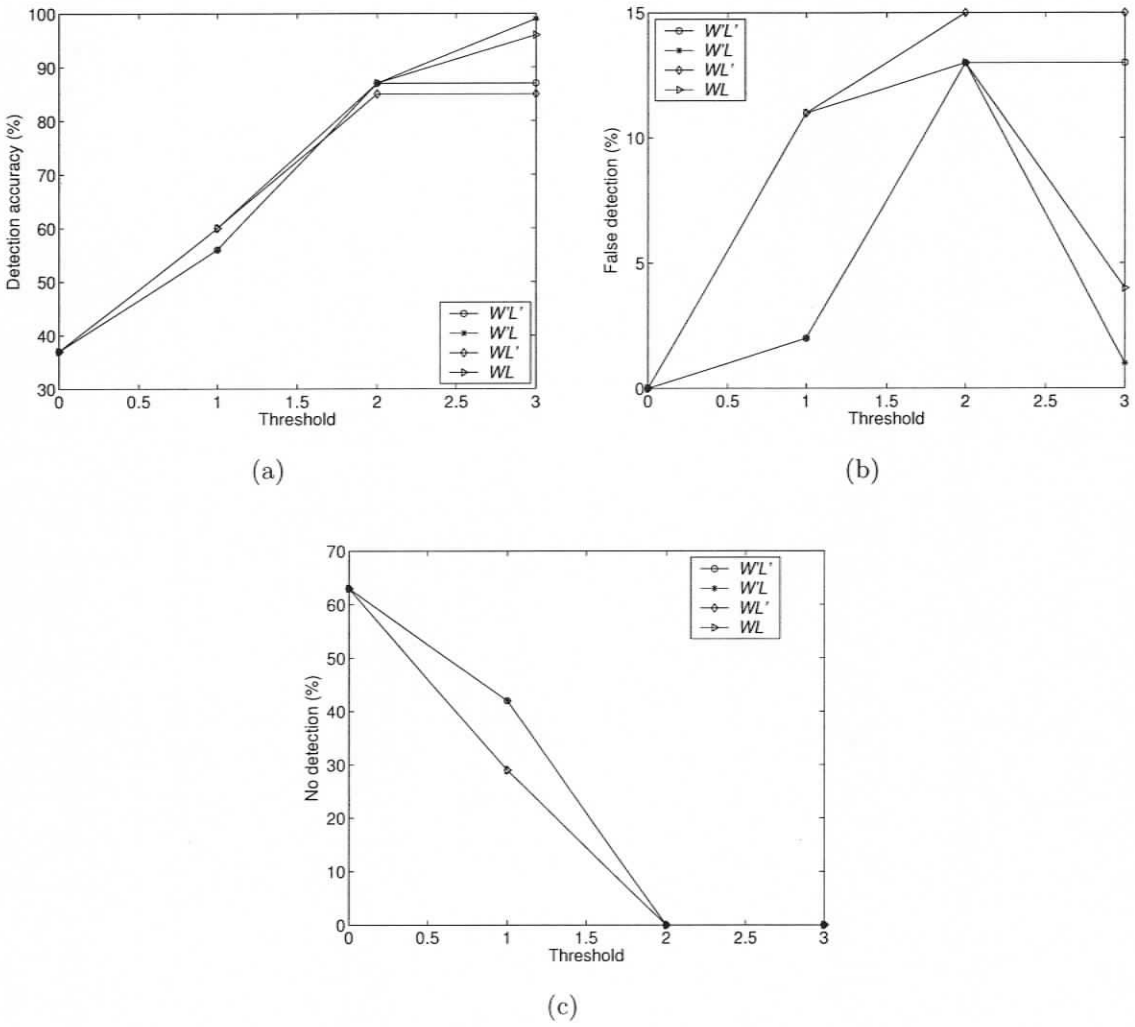


Figure 6.8: Experimental results on dataset$_1$ for R': (a) detection accuracy, (b) false detection, (c) no detection.

Table 6.5 for the meaning). However in general, R increases the detection accuracy and word boundary decreases the detection accuracy as in Figure 6.10.

Figure 6.12 and Figure 6.13 show the experimental results on dataset$_2$ for R' and R, respectively. These are to see the effects of L on the system performance. Like

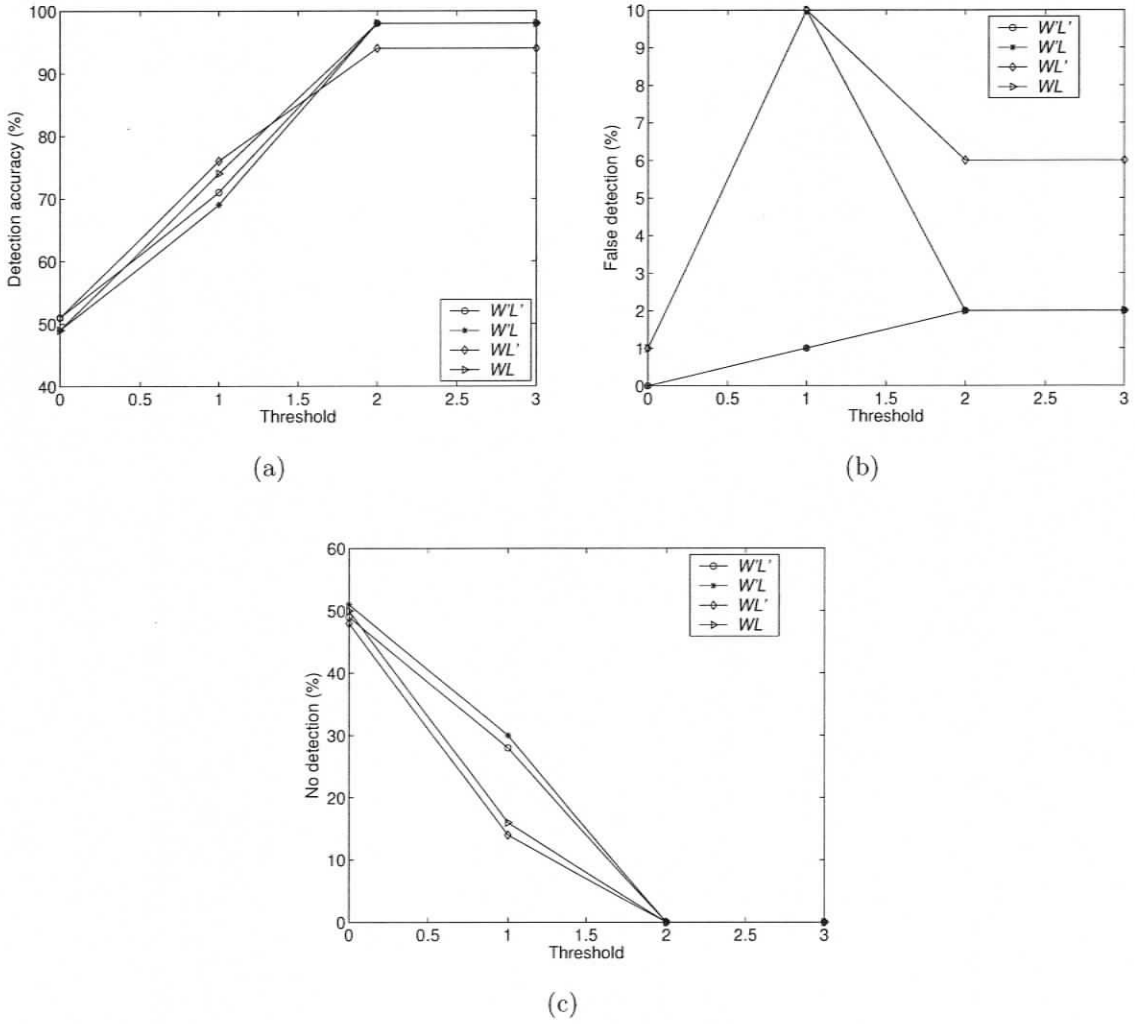


Figure 6.9: Experimental results on dataset$_1$ for R: (a) detection accuracy, (b) false detection, (c) no detection.

Figure 6.8 and Figure 6.9, both figures prove that longer match gives better results.

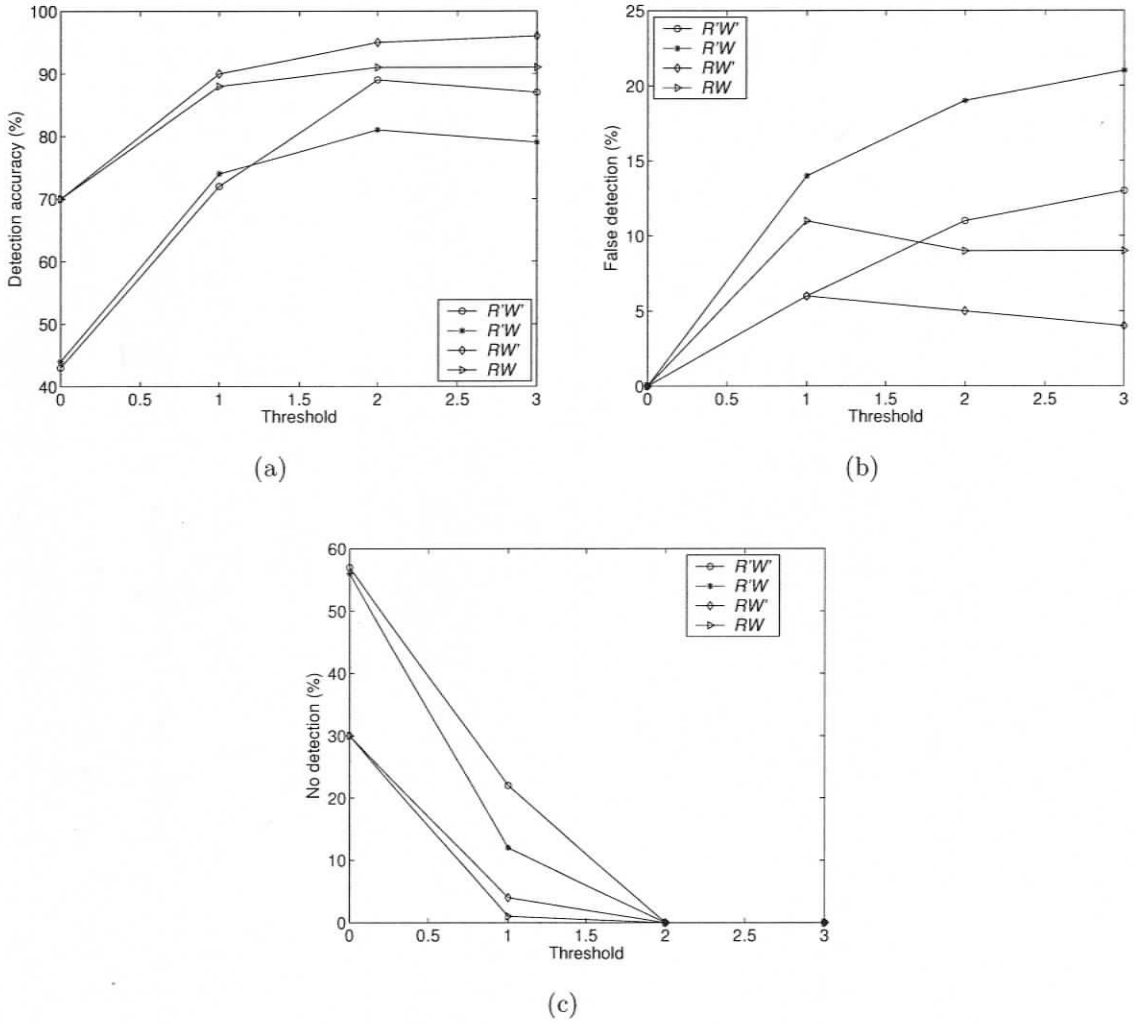


Figure 6.10: Experimental results on dataset$_2$ for L': (a) detection accuracy, (b) false detection, (c) no detection.

Experiment on Dataset$_3$

Like experiments on dataset$_1$ and dataset$_2$, $RW'L$ give better performance for dataset$_3$. But for the dataset$_3$, for $\Theta > 2$, performance is starting to degrade most of the cases. This is because that dataset$_3$ is built with the spam words from real spammers. The

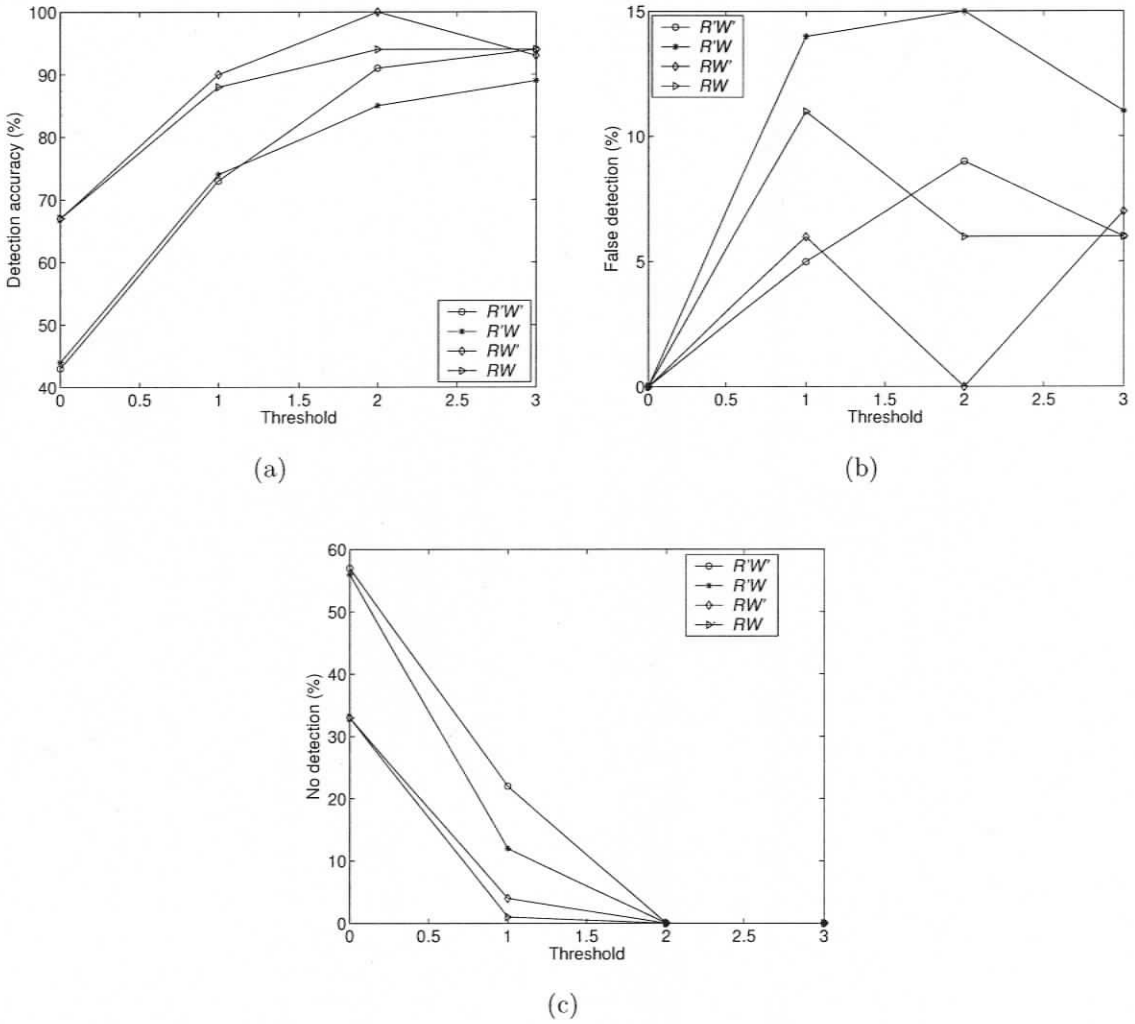


Figure 6.11: Experimental results on dataset$_2$ for L: (a) detection accuracy, (b) false detection, (c) no detection.

goal of spammers is to go through the spam filters and yet convey the message to the readers without much confusion. If the Levenshtein distance between an obfuscated word and the original word is greater than 2, the reader would most probably be confused with the obfuscated word. For this reason, experiments on dataset$_3$ gives

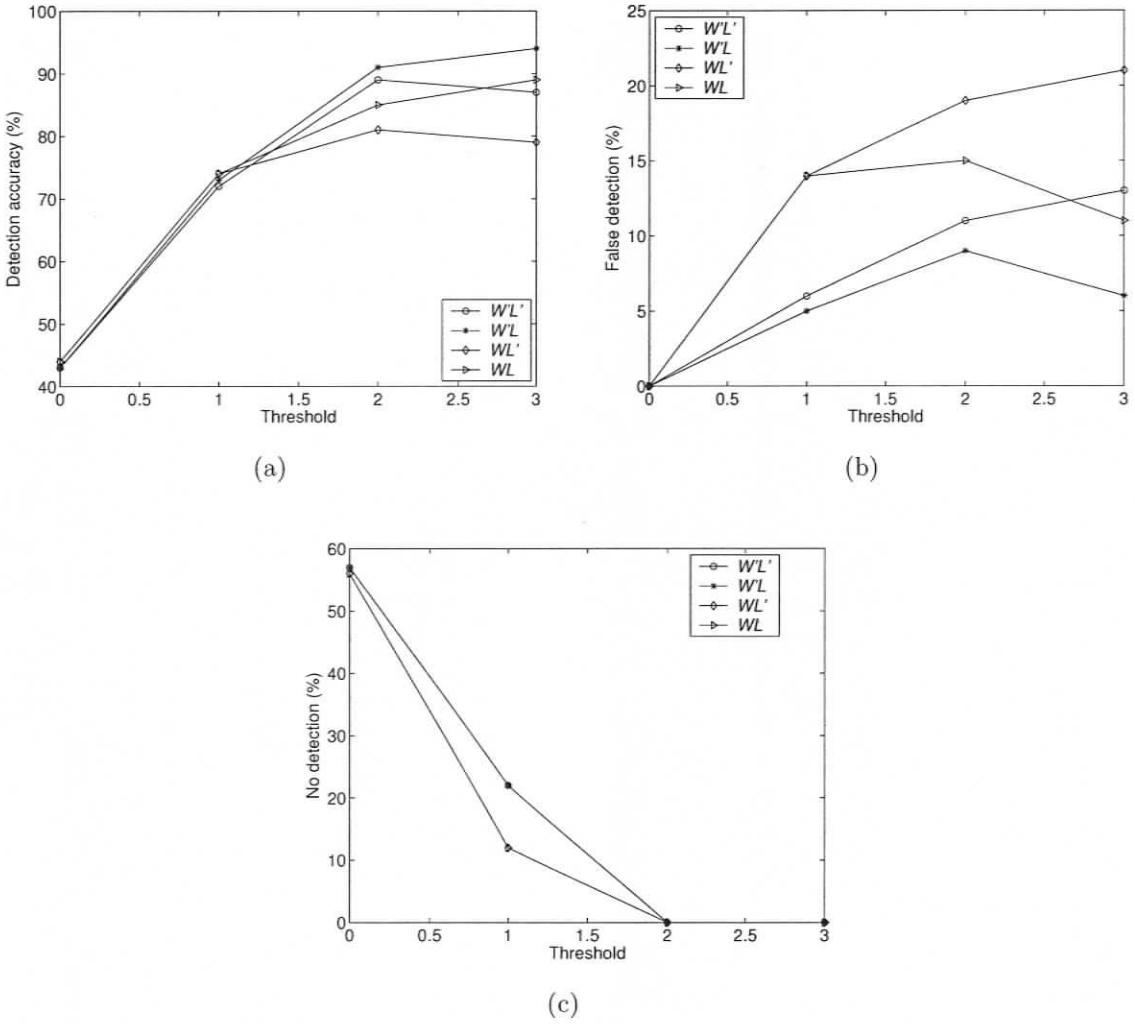


Figure 6.12: Experimental results on dataset$_2$ for R': (a) detection accuracy, (b) false detection, (c) no detection.

best detection performance at $\Theta = 2$.

From the above experiments, we can conclude that

- Θ gives the best results:
  - ▷ No detection becomes 0 when $\Theta \geq 2$.

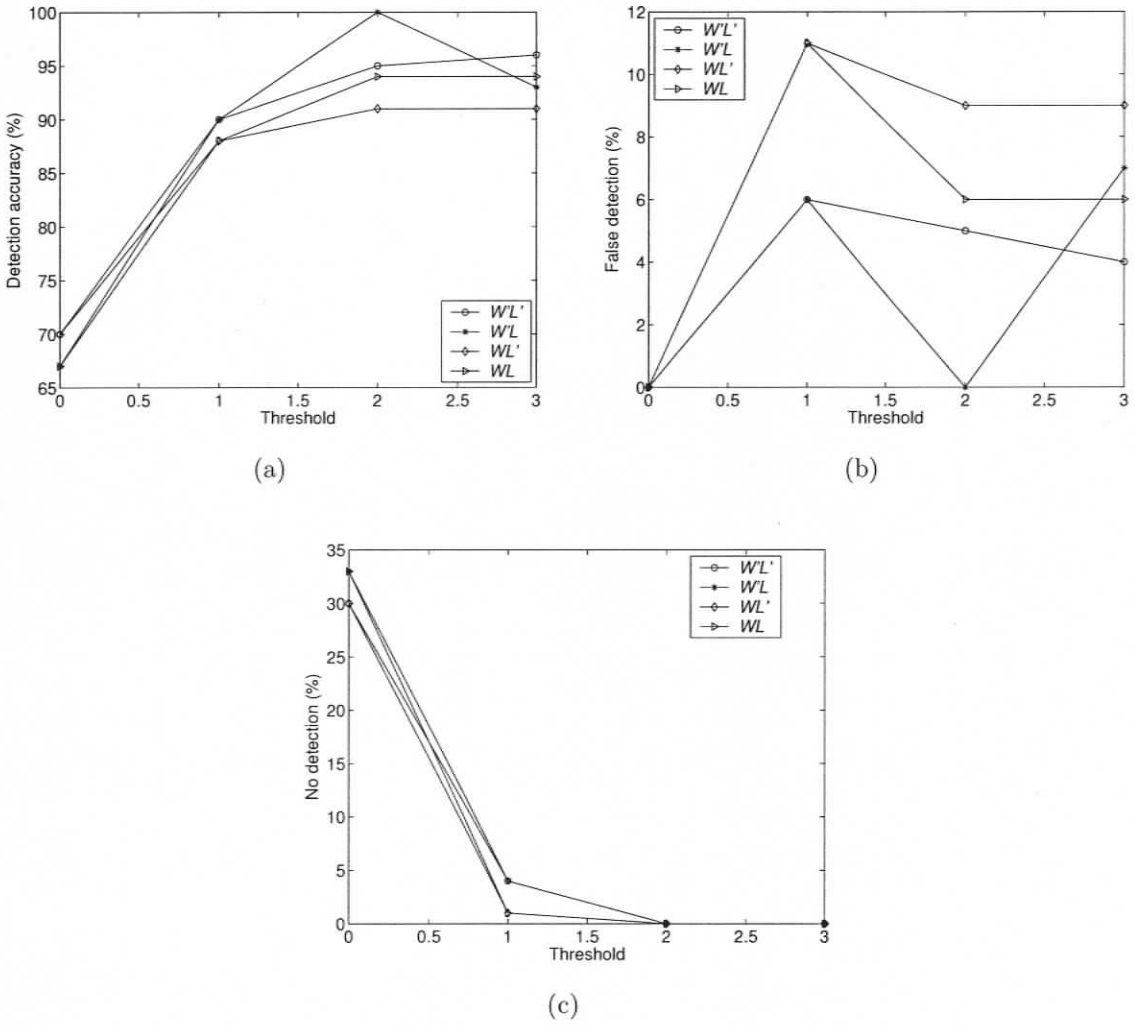


Figure 6.13: Experimental results on dataset$_2$ for R: (a) detection accuracy, (b) false detection, (c) no detection.

- ▷ Most of the case, especially the experiments with dataset$_3$, if $\Theta > 2$, the detection accuracy decreases due to higher false detection.

- R gives better result than that of R'.
- W' gives better result than that of W.

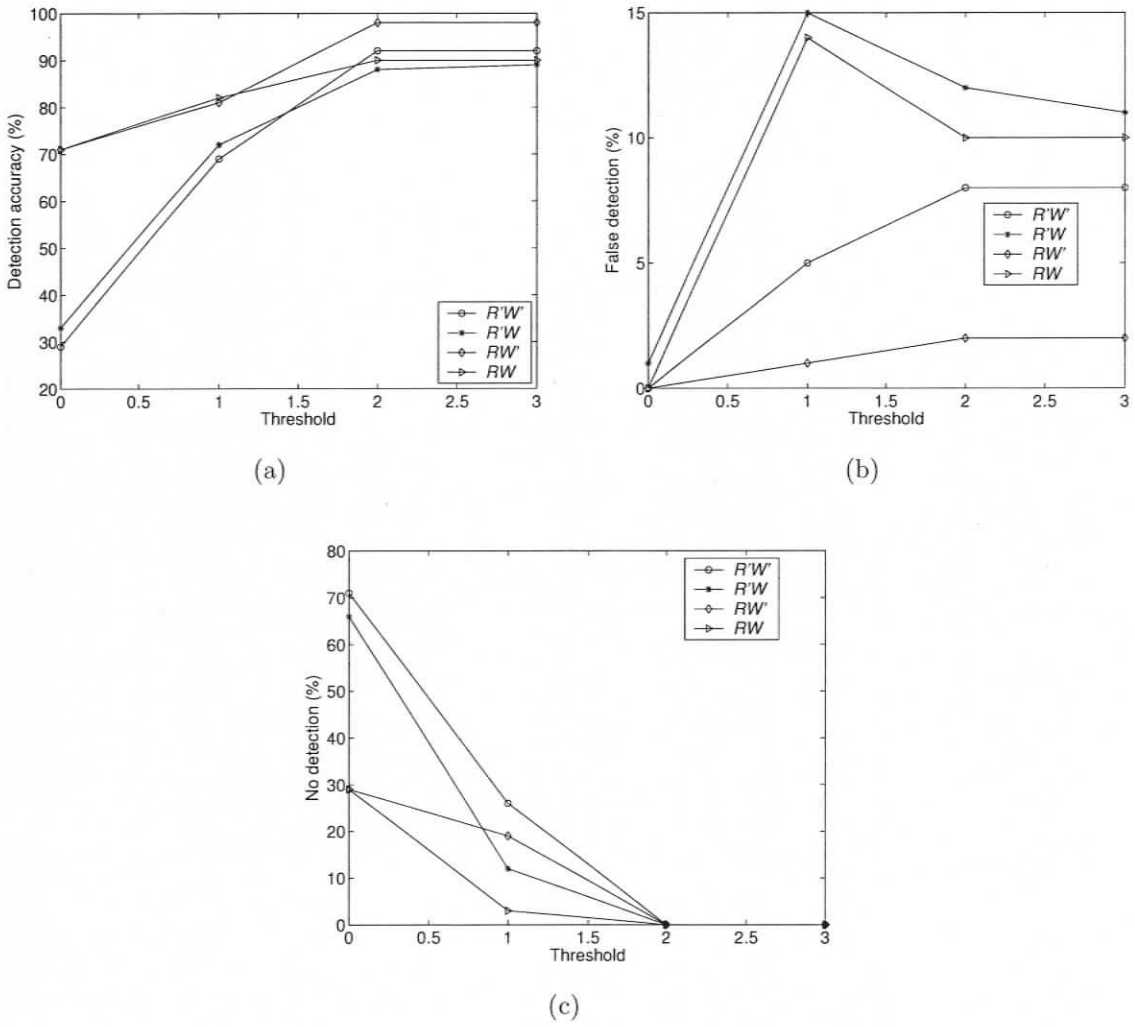


Figure 6.14: Experimental results on dataset$_3$ for L': (a) detection accuracy, (b) false detection, (c) no detection.

- L gives better result than that of L'.
- Maximum detection accuracy is 100% (on dataset$_2$, for $\Theta = 2$, RW'L).

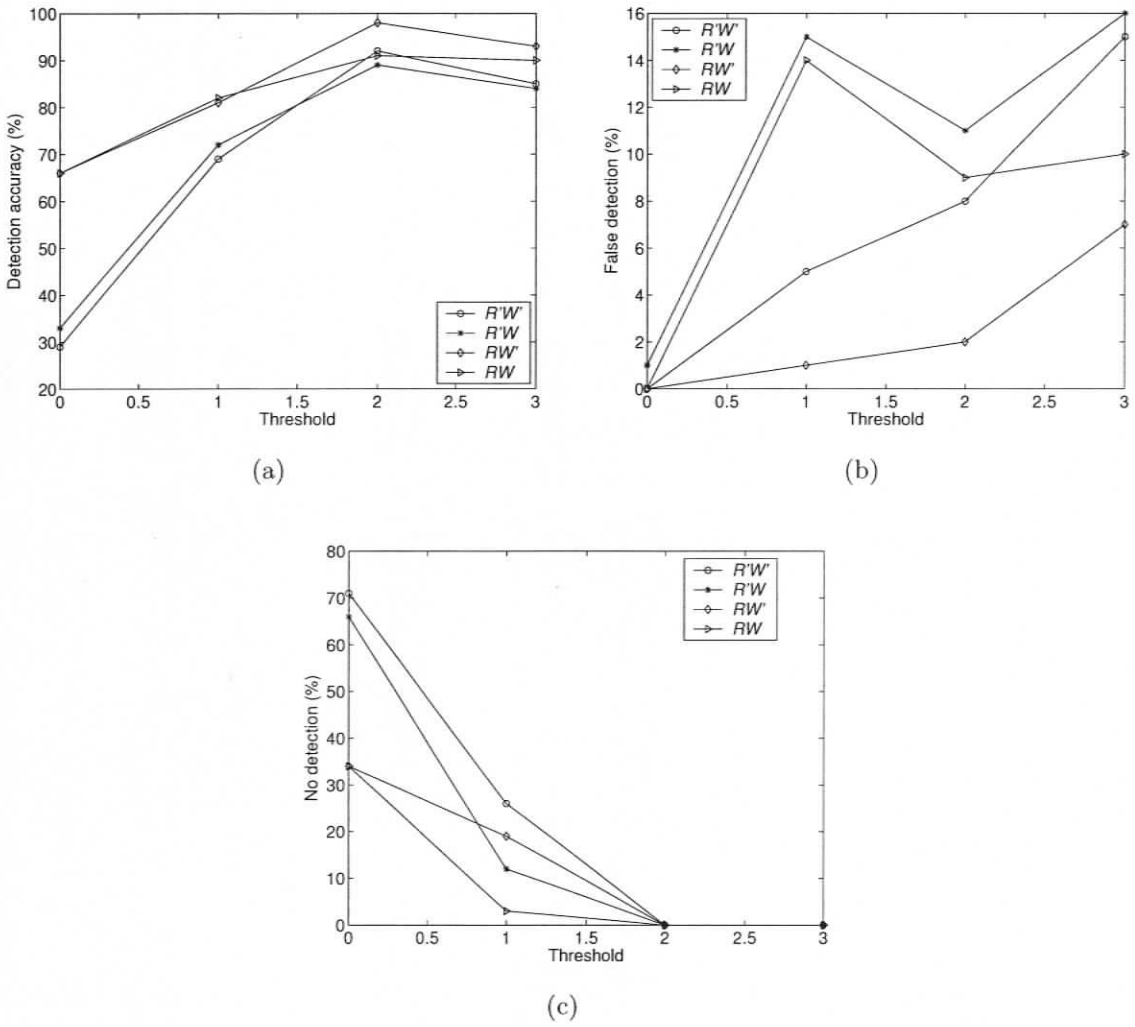


Figure 6.15: Experimental results on dataset$_3$ for L: (a) detection accuracy, (b) false detection, (c) no detection.

6.7.3 Time complexity

The time complexity for preprocessing is $\mathcal{O}(m)$, where m is the number of characters in the obfuscated word. Since the time complexity of the preprocessing is easy to understand, we don't further explain or experiment for the preprocessing time com-

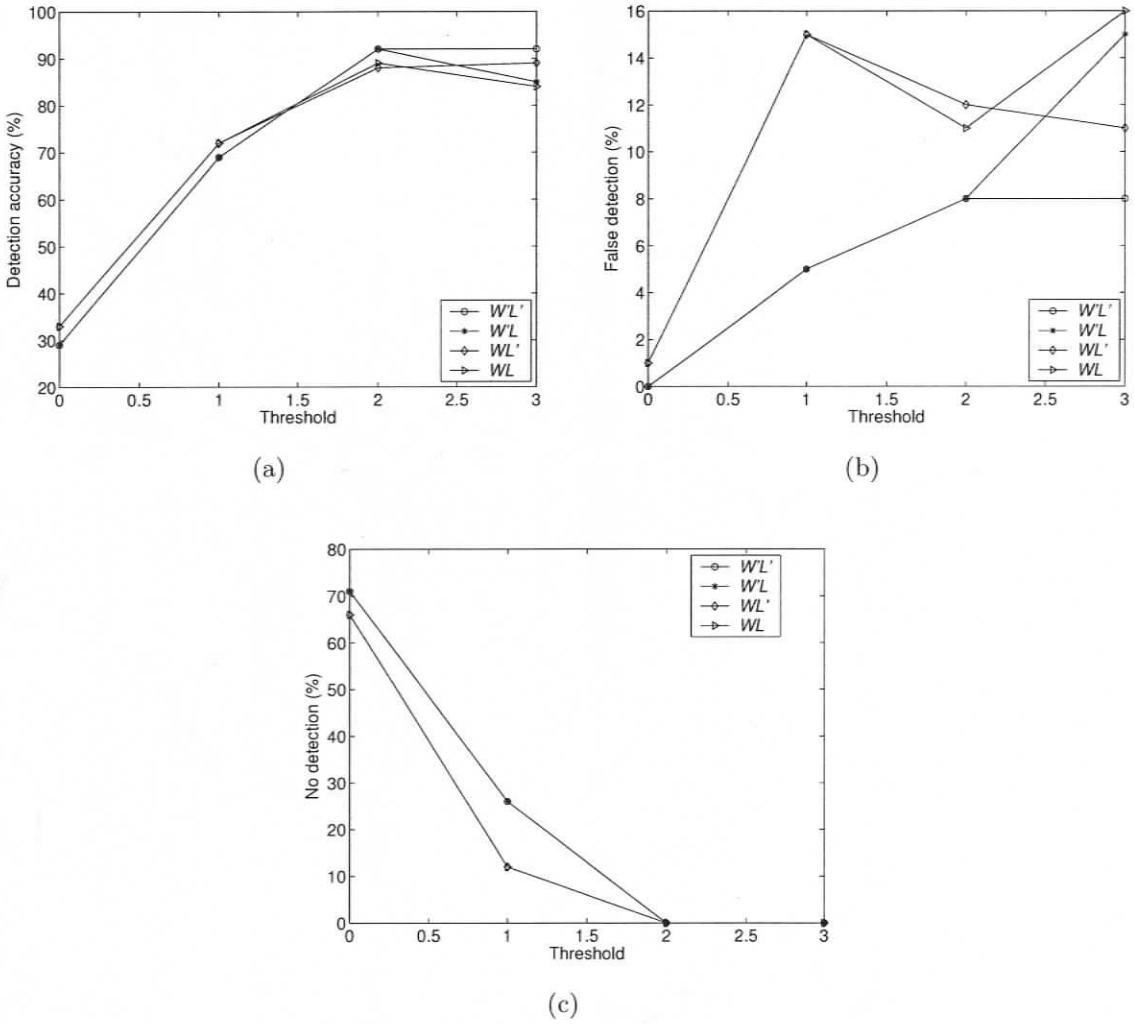


Figure 6.16: Experimental results on dataset$_3$ for R': (a) detection accuracy, (b) false detection, (c) no detection.

plexity. We are interested here to determine the time complexity of detecting the best-matched word in the automaton.

We have used only dataset$_3$ to determine the time complexity of our system. We assume that one time step is required to execute the dynamic program, i.e., the

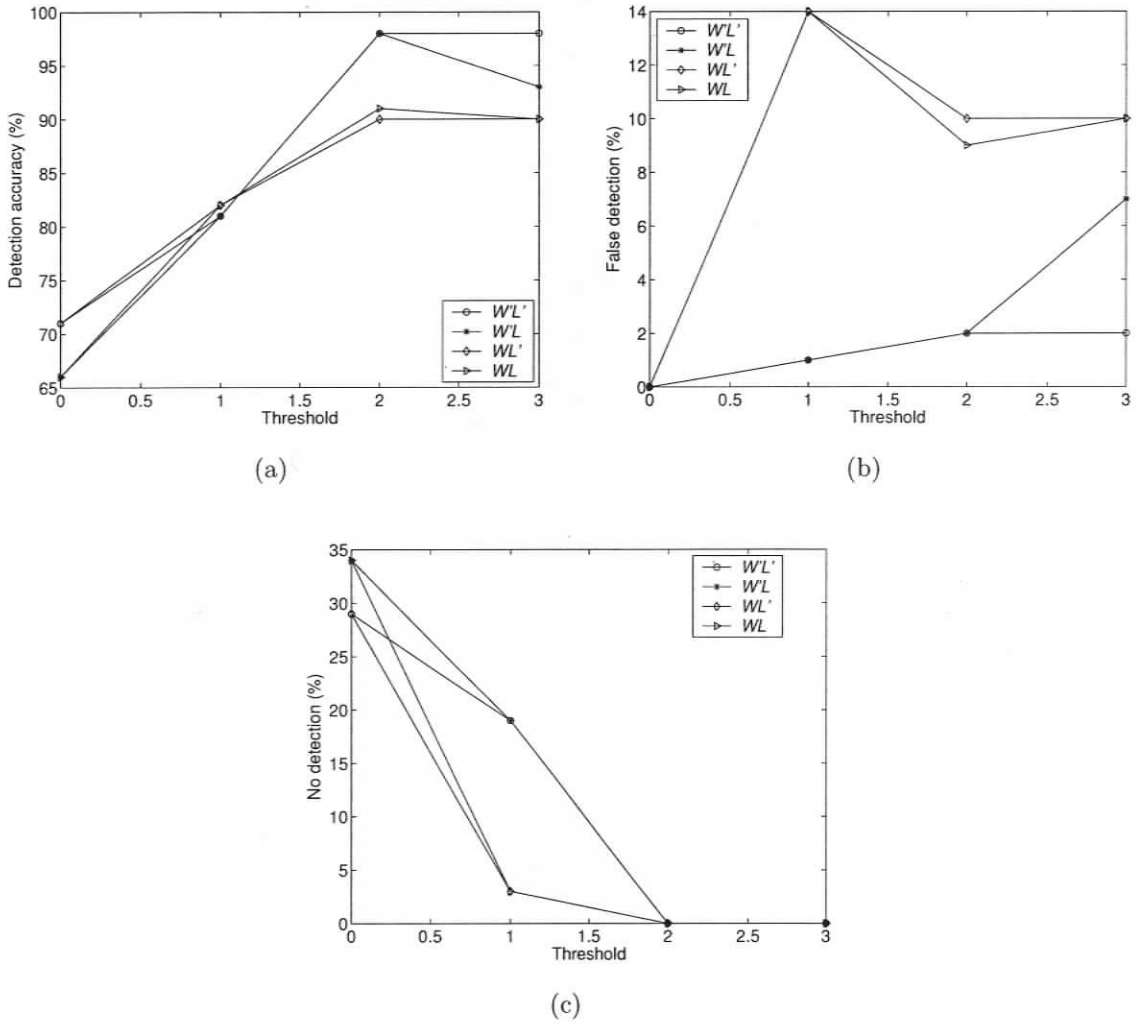


Figure 6.17: Experimental results on dataset$_3$ for R: (a) detection accuracy, (b) false detection, (c) no detection.

simulation requires one time step each time it visits a state of the whole simulation space. Thus to determine the time complexity, we count the number of visited states of the simulation space (not automaton states) during simulation. Among different variations of L, W, and R, we have chosen L', W', and R to determine our time

complexity.

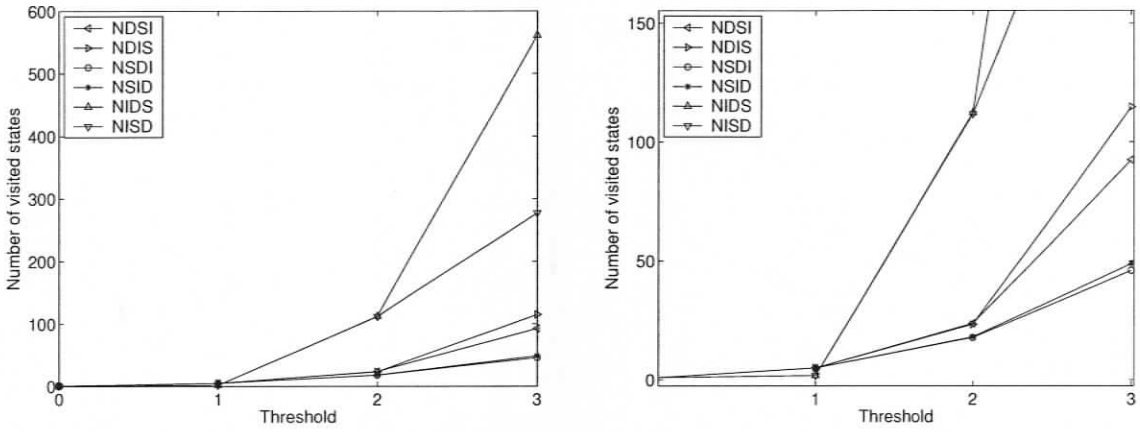


Figure 6.18: (a) Normal, (b) Zoomed between 0-150.

Figure 6.18 shows the experimental results for six variations of recursive calling order. In Section 6.6, we mentioned that the simulation program is recursively called four times for: normal (N), deletion (D), insertion (I), and substitution (S). Since for normal call, the cost remains the same, we prefer to put N at first in order. The remaining three calls can be ordered in six different ways. So, Figure 6.18 has six plot for each of these variations. Assume Ł is the number of characters in the preprocessed obfuscated word. For a meaningful value, we have normalized the number of visited states by Ł, i.e., the numbers in y-axis in Figure 6.18 must be multiplied by Ł to get the actual value. In Section 6.7.2, we proved that $\Theta = 2$ gives the best detection accuracy in most of the cases. In Figure 6.18, for $\Theta = 3$ increases time complexity of the system significantly. Thus $\Theta = 2$ is the best in respect of both time and accuracy. In Figure 6.18, the variation NSDI gives the best time complexity. To explain this, we have tabulated in Table 6.6 the counts of the occurrences of the different types of errors in dataset_3. Fillers are handled by the preprocessing. To calculate the

Table 6.6: Occurrence of different types of errors.

Types of errors	No. of occurrence	(%)
insertion (filler)	100	21.79
insertion (char)	100	21.79
deletion	12	2.61
transpose	5	1.09
substitution (filler)	164	35.73
substitution (char)	41	8.93
word boundary	37	8.06

time complexity of the dynamic programming, we have to exclude fillers from the calculation. Then insertion occurs more than deletion or substitution. According to the count, NISD should give the best time complexity. This does not happen during the simulation, because of the position of occurrences of errors. For example, "iagra" requires more time than that of "vigra", i.e., if the obfuscation occurs in the earlier position, it requires more time to get the result. From the experiments, the time complexity is much less than $n \times m$ which is required to simulate an NFA, where n = number of states in the NFA.

6.8 Complexity Analysis

To determine the time complexity of the dynamic programming, let us consider a tree-like structure as drawn in Figure 6.5. If $\Theta = 0$, then number of possible paths in that tree is 1. The number of possible paths for $\Theta = 1, 2, 3, \cdots, \text{Ł}$ are 3Ł, $3^2((\text{Ł} - 1) + \cdots + 1))$, $3^3((\text{Ł} - 2) + \cdots + 1) + (\text{Ł} - 3) + \cdots + 1) + (\text{Ł} - 4) + \cdots + 1))$, $\cdots$, $3^{\text{Ł}}$, respectively.

In Figure 6.5, there are four outgoing transitions from each state. To simplify our calculation, assume the probabilities for all these transitions are equal and the dynamic programming checks all four transitions with same priority. For each path in the tree in Figure 6.5, Ł states are visited. Then the average number of visited states, V_a can be written as,

- For $\Theta = 0$,

$$V_a = Ł^2$$

- For $\Theta = 1$,

$$V_a = 3Ł^2$$

- For $\Theta = 2$,

$$V_a = 3^2((Ł - 1) + \cdots + 1))Ł$$

- In general,

$$V_a = 3^{\Theta}Ł^2 \tag{6.2}$$

Comparing the theoretical value of V_a (Equation 6.8) with the practical value of V_a (Figure 6.18), we can conclude the following:

- V_a increases exponentially in both cases.
- Because of the state pruning of the dynamic programming, V_a in practical cases, would be smaller than theoretical V_a.
- Also in practical cases, transition probabilities are not equal and so the different sequences of transitions show different values of V_a in Figure 6.18.

6.9 Conclusion

In this chapter, we have proposed a novel Best Match technique required to detect best-matched English words of obfuscated spam words. We have used a nondeterministic finite automaton (NFA) to build the English dictionary. We have used dynamic programming with state pruning to detect the best-matched word of an obfuscated spam word in the NFA. We have done extensive numerical simulations to prove the accuracy of our proposed system. We have varied different design parameters during simulations to show the effects of these parameters on the accuracy of the detection. Our system can detect best-matched words of the words obfuscated by spammers using five different techniques: insertion, deletion, substitution, transpose, and word boundary. Upto our knowledge, no other system can deal with all these obfuscating techniques so quickly as ours. Using simulations, we have determined the time complexity of our system which is far less than the time complexity required to simulate an NFA. We also present the time complexity analysis that complies the simulation results.

Our proposed system cannot detect "VwwIwwAwwGwwRwwA" using any of the above mentioned variations. Because the Levenshtein distance between "VwwIwwAwwGwwRwwA" and "viagra" is 10 which is greater than the biggest $\Theta = 3$ in out experiments. We can introduce more intelligence inside the preprocessing to eliminate w's from "VwwIwwAwwGwwRwwA". We keep this as a future work of our research on Best Match.

Chapter 7

Conclusion and Future Works

This dissertation five major contributions.

7.1 Exact Match–Sequential Solution

Exact match is to find all occurrences of pattern in the text string. Exact match is required for differentiated service, server load balancing, security and so on. The existing sequential solutions have the following problems: (i) use pre-processing, i.e., not suitable for hardware implementation, (ii) use large amount of memory, and (iii) very complex for the implementation.

We have proposed an exact matching algorithm that requires reduced time complexity. The proposed algorithm requires a small amount of memory, and shows better performance than any other related algorithms as proved by numerical analysis and extensive computer simulations. These works are published in [55, 81].

7.2 Exact Match–Parallel Solution

The average time complexity for implementing the exact matching algorithm on a single processor was proved to be $\mathcal{O}(n)$. To meet the requirement of the exact matching algorithm, several hardware solutions were proposed that made use of advances in VLSI and processor array design techniques. The existing parallel solutions have the following problems: (i) use Ad-hoc approach and (ii) do not guarantee to exhibit optimum speed and area complexities.

We have presented a systematic technique for expressing the string search algorithm as a regular iterative expression to explore all possible processor arrays. 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. The proposed designs exhibit optimum speed and area complexities. These works are published in [65, 82, 83].

7.3 Embedding Technique

The main disadvantage of existing methodologies on parallel exact match is that different-sized processor arrays are produced each time the algorithm parameters change. Also the resulting processor array size might exceed the hardware resource constraints. The parallel solution requires an embedding technique that embeds a source processor array onto a target processor array having smaller number of PEs to meet the hardware resource constraint. In the existing techniques, the embedded architectures shows the degraded performance.

We have proposed a novel embedding technique. Through numerical analysis and extensive computer simulation, it is proved that the performance of the embedded

array shows the same performance as the source array. These works are published in [66, 84].

7.4 Longest Prefix Match

LPM is primarily used to determine the best next_hop route for a packet based on its destination address. The existing solutions have the following problems: (i) expansion problem, and (ii) slow update of the lookup-table.

We have proposed a novel variable-stride multi-bit trie data structure for IP-lookup table to assist fast IP-lookup and fast lookup table update. In this dissertation, we first explicitly elaborate the solution of a problem in expanding IP addresses. Through extensive computer simulation on several routing tables, it is proved that our proposed algorithm shows better performance (lookup and update time) than existing algorithms. However, our proposed technique requires larger memory than others. But the memory requirement is quite acceptable considering the current memory availability and price.

In future, we plan to propose a hardware for the LPM problem described in Chapter 5.

7.5 Best Match

As people are getting and blocking spams, the spammers are becoming cunning day by day. They obfuscate the well-known spam words in different ways to circumvent the spam filtering strategies. If obfuscated words can be replaced by the best-matched English words before filtering, then the filtering techniques would perform better. The existing solutions do not deal with five techniques used by spammers: (i) insertion, (ii) deletion, (iii) transpose, (iv) substitution, and (v) word boundary.

We have proposed a novel Best Match technique required to detect best-matched English words of obfuscated spam words. We have used an NFA to build the English dictionary. We have used dynamic programming with state pruning to detect the best-matched word of an obfuscated spam word in the NFA. We have done extensive numerical simulations to prove the accuracy of our proposed system. Upto our knowledge, no other system can deal with all these obfuscating techniques so quickly as ours. These works are published in [87].

In future, we plan to:

- introduce more intelligence inside the preprocessing of the Best Match technique described in Chapter 6, and
- propose a hardware for the Best Match technique described in Chapter 6.

Bibliography

[1] A. Deb, "What is a network processor?" in *A presentation to Networld Interop 2000.* Vitesse Semiconductor Corporation, May 2000.

[2] D. Husak. (2000, May) Network processors: A definiiton and comparison. M957198397651.pdf. [Online]. Available: www.motorola.com/collateral/

[3] T. Wolf, "Design of an instruction set for modular network processors," IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC 21865 (98398), October 2000.

[4] N. Shah, "Understanding network processors," Master's thesis, University of California, Berkeley, CA 94720, USA, September 2001.

[5] (2003, Jan.) Architecture guide: C-5e/C-3e network processor. C5EC3EARCH-RM.pdf. [Online]. Available: http://e-www.motorola.com/brdata/PDFDB/docs/

[6] (2002) Intel IXP2800 network processor. 279054.htm. [Online]. Available: http://www.intel.com/design/network/prodbrf/

[7] (2002, February) IBM PowerNP NP4GS3 network processor. NP4GS3_ds_public.11.pdf. [Online]. Available: http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF7%785256983006A3809

[8] P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk, Eds., *Network Processor Design–Issues and Pactices.* 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205: Morgan Kaufmann Publishers, October 2002, vol. 1.

[9] (2003, Feb.) NP-1 family–the world's most highly integrated 10-Gigabit 7-layer network processors. in_prod.html. [Online]. Available: http://www.ezchip.com/html/

[10] (2002, November) 10G network processor chip set (APP7 50NP and APP750TM). PB01137-2.pdf. [Online]. Available: http://www.agere.com/enterprise_metro_access/docs/

[11] (2002, September) Advanced payloadplus network processor (APP550). PB02024-2.pdf. [Online]. Available: http://www.agere.com/enterprise_metro_access/docs/

[12] (2003, Feb.) Cisco 7200 series network processing engine NPE-G1. npeg1_wp.pdf. [Online]. Available: http://www.cisco.com/warp/public/cc/pd/ifaa/prossor/prodlit/

[13] (2001) ClassiPI network classification processor:PM2329. index.html. [Online]. Available: http://www.pmc-sierra.com/products/details/pm2329/

[14] (2002) A high-performance/low-power MIPS (SOC) with PCI controller. 26329C_Alchemy_Au1500.pdf. [Online]. Available: http://www.amd.com/us-en/assets/content_type/DownloadableAssets/

[15] (2002) nP7510–10 Gbps network processor. [Online]. Available: http://www.amcc.com/cardiff/docManagement/displayProductSummary.jsp?pro%dId=nP7510

[16] L. Wirbel, "Network processor handles mix of carrier services," *EE Times*, September 2000.

[17] (2002) Internetworking processor (InP) family overview. products.html. [Online]. Available: http://www.baymicrosystems.com/solutions/

[18] (2003, Feb.) Multi-service processor architecture. prod_tech.html. [Online]. Available: http://www.brecis.com/

[19] (2002) BCM 1250–product brief. BCM1250.pdf. [Online]. Available: http://www.broadcom.com/pbs/

[20] (2002) A wolf in MIPS clothing: Clearwater's multi-thread packet processor handles layer 4-7 tasks OC-192 speeds. prod200.html. [Online]. Available: http://www.chipcenter.com/networking/products_200-299/

[21] (2002) Sneak preview–hexi-deci-clops: Cognigine's silicon monster uses 16 reconfigurable processors to devour OC-192 traffic. prod201.html. [Online]. Available: http://www.chipcenter.com/networking/products_200-299/

[22] (2002) CX27470–traffic stream processor. [Online]. Available: http://web2.mindspeed.com/default.sph/SaServletEngine.class/Web/product%s/index.jsp?catalog_id=203&cookietrail=0,8,49

[23] L. Gwennap, "Lexra offers netvortex net processor as licensable core," *EE Times*, June 2000.

[24] (2003, Feb.) IQ2200 network processor. [Online]. Available: http://www.vitesse.com/products/documents.cfm?document_id=321&family_id%=5

[25] (2002) Xelerator network processors–X10s, X10d, X10q. 44.pdf. [Online]. Available: http://216.146.236.120/uploads/files/

[26] P. Mehrotra and P. D. Franzon, "Novel hardware architecture for fast address lookups," *IEEE Communications Magazine*, vol. 40, no. 11, pp. 66–71, Nov. 2002.

[27] M. Nossik. (2002) Optimizing network processing with deep packet classification. OPTIMIZING_WP.pdf. [Online]. Available: http://www.idt.com/docs/

[28] ——. (2002) The relationship between classification processors and network search engines. RELATIONSHIP_WP.pdf. [Online]. Available: http://www.idt.com/docs/

[29] ——. (2002, July) The challenges associated with implementing a complete classification solution. 92009-WP-1.1.1Challenges%20White%20Paper.pdf. [Online]. Available: http://www.solidum.com/resource_center/pdfs

[30] F. Welfeld. (2001) Packet classification processor: The ultimate branching machine. 92006-WP-1.1.1State%20Machine%20White%20Paper.pdf. [Online]. Available: http://www.solidum.com/resource_center/pdfs

[31] ——. (2002) The case for a standalone classification processor. CASE_WP_16548.pdf. [Online]. Available: http://www.idt.com/docs/

[32] D. Husak and R. Gohn. (2001, May) Network processor programming models: The key to achieving faster time-to-market and extending product life. CPPM00WP100.pdf. [Online]. Available: http://e-www.motorola.com/collateral/

[33] M. Nossik. (2002) Optimizing network processing with deep packet classification. OPTIMIZING_WP.pdf. [Online]. Available: http://www.idt.com/docs/

[34] (2000, April) Scaling next generation web infrastructure with content-intelligent switching. l7_white_paper1.pdf. 50 Great Oaks Boulevard, San Jose, California 95119. [Online]. Available: www.nortelnetworks.com/products/library/collateral/intel_int/

[35] A. Arnaud, "Deep packet classification coprocessors," in *Network Processors Conference*. Raqia Networks, Inc., October 2001.

[36] F. Welfeld, "Network processing in content inspection applications," *ISSS*, pp. 197–201, October 2001.

[37] T. Lakshman and D. Stiliadis, "Highspeed policybased packet forwarding using efficient multidimensional range matching," bell Laboratories.

[38] 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, March/April 2001.

[39] (2001, Feb.) Toward content-based classification. pdf. [Online]. Available: http://www.pmc-sierra.com/cgi-bin/download_p.pl?res_id=1619&filename=20%02233.pdf

[40] (2003) StoneGate high availability firewall and vpn. html. [Online]. Available: http://www.stonesoft.com/products/StoneGate/

[41] A. S. Tanenbaum, *Computer Networks*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.

[42] *IA-32 Intel Architecture Software Developer's manual–Volume 1: Basic Architecture*, 245472-011, Intel Corporation, Mt. Prospect, IL, 2003.

[43] T. Chu. (2002) Network processors and coprocessors for broadband network applications. icd-submission-short.PDF. [Online]. Available: http//web73149.ntx.net/Technology/

[44] S. Nilsson and G. Karlsson, "Ip-address lookup using lc-tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083–1092, June 1999.

[45] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proceedings of IEEE INFOCOM'98*, vol. 3, San Francisco, CA, Mar-Apr 1998, pp. 1240–1247.

[46] N. F. Huang and S. M. Zhao, "A novel IP-routing lookup scheme and hardware architecture for multigigabit switching routers," *IEEE journal on selected areas in communications*, vol. 17, no. 6, pp. 1093–1104, June 1999.

[47] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," in *Proceedings of IEEE INFOCOM'98*, vol. 3, San Francisco, CA, Mar-Apr 1998, pp. 1248–1256.

[48] D. Bursky, "Search engines take on larger forwarding tables," *Electronic Design*, vol. 51, no. 27, p. 48, Dec. 2003.

[49] K. Pagiamtzis, "Content-addressable memory introduction, [online document]," [Dec 21, 2003] Available at: http://www.eecg.toronto.edu/ pagiamt/cam/camintro.html.

[50] H. Liu, "Reducing routing table size using ternary-cam," in *The Ninth Symposium on High Performance Interconnects.* Stanford, California, USA: IEEE, August 2001, pp. 69–74.

[51] M. Peng and S. Azgomi, "Content-addressable memory (cam) and its network applications, [online document]," [Dec 21, 2003] Available at: http://www.eetasia.com/ARTICLES/2000MAY/2000MAY03_MEM_NTEK_TAC.PDF.

[52] "Cams (content-addressable memory), [online document]," [Dec 21, 2003] Available at:http://e-www.motorola.com/webapp/sps/site/taxonomy.jsp?nodeId=0154248624.

[53] M. Peyravian, G. Davis, and J. Calvignac, "Search engine implications for network processor efficiency," *IEEE Network*, vol. 17, no. 4, pp. 12–14, July–August 2003.

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

[55] A. N. M. E. Rafiq, M. W. El-Kharashi, and F. Gebali, "A fast string search algorithm for deep packet classification," *Computer Communications*, vol. 27, no. 15, pp. 1524–1538, Sept. 2004.

[56] T. Lecroq, "Experiments on string matching in memory structures," *Software: Practice and Experience*, vol. 28, no. 5, pp. 561–568, Apr. 1998.

[57] H. T. Kung and C. E. Leiserson, "Systolic arrays for VLSI," in *Sparse Matrix Symposium*. Philadelphia, PA: Society of Industrial and Applied Mathematicians, 1978, pp. 256–282.

[58] S. K. Rao and T. Kailath, "Regular iterative algorithms and their implementation on processor arrays," *Proceedings of the IEEE*, vol. 76, no. 3, pp. 259–269, Mar. 1988.

[59] S. Y. Kung, *VLSI array processors*, ser. Information and system sciences, T. Kailath, Ed. Prentice Hall, 1988.

[60] E. M. M. Abdel-Raheem, "Design and VLSI implementation of multirate filter banks," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Victoria, 1995.

[61] E. Abdel-Raheem, F. El-Guibaly, and A. Antoniou, "Systolic implementation of FIR decimators and interpolators," *IEE Proceedings on Circuits Device System*, vol. 141, pp. 489–492, Dec. 1994.

[62] M. O. Esonu, A. J. Alkhalili, S. Hariri, and D. Al-Khalili, "Systolic arrays – how to choose them," in *IEE Proceedings-E Computers and Digital Techniques*, vol. 139, no. 3, May 1992, pp. 179–188.

[63] J. M. D. Y. Wong, "Optimization of computation time for systolic arrays," *IEEE Transactions on Computers*, vol. 41, no. 2, pp. 159–177, February 1992.

[64] F. El-Guibaly and A. Tawfik, "Mapping 3-d IIR digital filter onto systolic arrays," *Multidimensional Systems and Signal Processing*, vol. 7, no. 1, pp. 7–26, Jan. 1996.

[65] F. Gebali and A. N. M. E. Rafiq, "Processor array architectures for deep packet classification," *IEEE Transaction on Parallel and Distributed Systems*, vol. 17, no. 3, pp. 241–252, Mar. 2006.

[66] A. N. M. E. Rafiq and F. Gebali, "Embedding the systolic arrays for deep packet classification into a pre-existing processor array," in *Proceedings of the First International Computer Engineering Conference*, Cairo, Egypt, Dec. 2004, pp. 605–610.

[67] A. S. Tanenbaum, *Computer Networks*, 4th ed. Prentice Hall, 2003.

[68] B. Whitworth and E. Whitworth, "Spam and the social-technical gap," *IEEE Computer*, vol. 37, no. 10, pp. 38–45, Oct. 2004.

[69] "An overview of spam blocking techniques, [online document]," white paper, Barracuda Networks, available at: http://www.barracudanetworks.com/resources/docs/Spam_Techniques.pdf, Dec. 2004.

[70] J. Goodman, D. Heckerman, and R. Rounthwaite, "Stopping spam,," *Scientific American*, pp. 42–49, Apr. 2005.

[71] L. H. Gomes, C. Cazita, J. M. Almeida, V. Almeida, and J. W. Meira, "Characterizing a spam traffic," in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement (ICM)*, Taormina, Sicily, Italy, 2004, pp. 356–369.

[72] (2005, Feb.) Study: Spam costing companies $22 billion a year. html. [Online]. Available: http://www.cnn.com/2005/TECH/02/03/junk.email.ap/

[73] J. Lyman. (2006, Mar.) Spam costs $20 billion each year in lost productivity. html. [Online]. Available: http://www.linuxinsider.com/story/32478.html

[74] F. Garcia, J. Hoepman, and J. van Nieuwenhuizen., "Spam filter analysis," University of Nijmegen, the Netherlands, Tech. Rep. cs.CR/0402046v1, Feb. 2004.

[75] P. Graham. (2006, Mar.) Better bayesian filtering. html. [Online]. Available: http://www.paulgraham.com/better.html

[76] X. Wang, H. Duan, T. Q. Anh, and X. Li, "Dynamic rules' score adjustment in spam filter using users' feedback," in *Proceedings of the Fourth International Conference on Machine Learning and Cybernetics*, vol. 2, Guangzhou, China, Aug. 2005, pp. 665–669.

[77] H. Drucker, D. Wu, and V. N. Vapnik, "Support vector machines for spam categorization," *IEEE Transactions on Neural Networks*, vol. 10, no. 5, pp. 1048–1054, Sept. 1999.

[78] (2006, Mar.) Us spam crackdown shows mixed results. htm. [Online]. Available: http://news.zdnet.co.uk/internet/0,39020369,39229171,00.htm

[79] H. Lee and A. Ng, "Spam deobfuscation using a hidden markov model," in *Proceedings of the Second Conference on Email and Anti-Spam*, Stanford University, USA, July 2005.

[80] K. Kukich, "Techniques for automatically correcting words in text," *ACM Computing Surveys*, vol. 24, no. 4, pp. 377–440, 1992.

[81] A. N. M. E. Rafiq, M. W. El-Kharashi, and F. Gebali, "A fast string search algorithm for computer networking," in *2003 IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, F. Gebali, Ed., vol. 2, Victoria, BC, Canada, Aug. 2003, pp. 764–767.

[82] A. N. M. E. Rafiq, F. Gebali, and M. W. El-Kharashi, "A systolic array structure for string searching," in *Proceedings of the 2004 International Conference on Electrical, Electronic, and Computer Engineering (ICEEC'04)*, A. Wahdan, A. Amar, H. Fikry, and A. Salem, Eds., Ain Sams University, Cairo, Egypt, Sept. 2004, pp. 281–284.

[83] A. N. M. E. Rafiq and F. Gebali, "Processor array design for deep packet classification," in *Proceedings of the Fourth IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, Dec. 2004, pp. 81–83.

[84] ——, "A novel technique for embedding a processor array into a smaller array," *IEEE Transactions on Parallel and Distributed Systems (submitted)*, 2006.

[85] ——, "A novel data structure for fast ip-lookup and fast update," *IEEE/ACM Transactions on Networking (submitted)*, 2006.

[86] A. N. M. E. Rafiq, M. N. Marsono, F. Gebali, and M. W. El-Kharashi, "Identifying obfuscated spam words," *IEEE Transactions on Knowledge and Data Engineering (submitted)*, 2006.

[87] A. N. M. E. Rafiq, M. W. El-Kharashi, and F. Gebali, "Systolic array-based string matching unit for spam blocking," in *Proceedings of The Fifth International Workshop System-on-Chip for Real-Time Applications*, W. Badawy and K. Iniewski, Eds., Banff, Alberta, Canada, July 2005, pp. 444–449.

[88] R. N. Horspool, "Horspool string searching algorithm, [online document]," [May 15, 2002] Available at: http://www-igm.uvic-mlv.fr/~lecrog/string, May 2002.

[89] M. Crochemore and C. Hancart, "Automata for matching patterns," in *Handbook of Formal languages*, G. Rozenberg and A. Salomaa, Eds. Springer-Verlag, Berlin, 1997, vol. 2, ch. 9, pp. 399–462.

[90] I. Simon, "String matching algorithms and automata," in *Proceedings of 1st American Workshop on String Processing*, R. Baeza-Yates and N. Ziviani, Eds., Universidale Federal de Minas Gerais, Brazil, 1993, pp. 151–157.

[91] T. Lecroq, "A variation on the Boyer-Moore algorithm," *Theoretical Computer Science*, vol. 92, no. 1, pp. 119–144, 1992.

[92] ——, "Recherches de mot," Ph.D. dissertation, University of Orléans, France, 1992.

[93] ——, "Experimental results on string matching algorithms," *Software–Parctice and Experience*, vol. 25, no. 7, pp. 727–765, 1995.

[94] G. Navarro and M. Raffinot, "A bit-parallel aproach to suffix automata: Fast extended string matching," in *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, Springer-Verlag, Berlin, 1998, pp. 14–31.

[95] C. Allauzen, M. Crochemor, and M. Raffinot, "Factor Oracle: a new structure for pattern matching," in *Proceedings of SOFSEM'99, Theory and Practice of Informatics*, J. Pavelka, G. Tel, and M. Bartosek, Eds., Springer-Verlag, Berlin, 1998, pp. 291–306.

[96] M. Crochemore and W. Rytter, "Text algorithms," Oxford University Press, 1994.

[97] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.

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

[99] T. Raita, "Tuning the Boyer-Moore-Horspool string searching algorithm," *Software–Practical and Experience*, vol. 22, no. 10, pp. 879–884, 1992.

[100] Z. Galil and J. Seiferas, "Time-space optimal string matching," *Journal of Computer and System Science*, vol. 26, no. 3, pp. 280–294, 1983.

[101] M. Crochemore and D. Perrin, "Two-way string matching," *Journal of the ACM*, vol. 38, no. 3, pp. 651–675, 1991.

[102] C. Charras, T. Lecroq, and J. D. Pehoushek, "A very fast string matching algorithm for small alphabets and long patterns," in *Proceedings of the 9th Annual Symposium on Combinaorial Pattern matching*, M. Farach-Colton, Ed., Springer-Verlag, Berlin, 1998, pp. 55–64.

[103] J. H. M. (jr) and V. R. Pratt, "A linear pattern-matching algorithm." University of California, Berkeley, Tech. Rep. 40, 1970.

[104] D. E. Knuth, J. H. M. (jr), and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal of Computing*, vol. 6, no. 1, pp. 323–350, 1977.

[105] L. Colussi, "Correctness and efficiency of the pattern matching algorithms," *Information and Computation*, vol. 95, no. 2, pp. 225–251, 1991.

[106] Z. Galil and R. Giancarlo, "On the exact complexity of string matching," *SIAM Journal on Computing*, vol. 21, no. 3, pp. 407–437, 1992.

[107] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, Oct. 1977.

[108] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Mountain View, CA, USA: Prentice-Hall, 1999, ch. String Matching, pp. 876–883.

[109] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Deux méthods pour accélérer l'algorithme de Boyer-Moore," in *Théorie des Automates et Applications, Actes des* 2^e *Journées Franco-Belges*, D. Krob, Ed., PUR 176, Rouen, France, 1991, pp. 45–63.

[110] A. Apostolico and R. Giancarlo, "The Boyer-Moore-Galil string searching strategies revisited," *SIAM Journal on Computing*, vol. 15, no. 1, pp. 98–105, 1986.

[111] L. Colussi, "Fastest pattern matching in strings," *Journal of Algorithms*, vol. 16, no. 2, pp. 163–189, 1994.

[112] R. N. Horspool, "Practical fast searching in strings," *Software–Practical and Experience*, vol. 10, no. 6, pp. 727–765, 1980.

[113] D. M. Sunday, "A very fast substring search algorithm," *Communications of the ACM*, vol. 33, no. 8, pp. 132–142, 1990.

[114] A. Hume and D. M. Sunday, "Fast string searching," *Software–Practical and Experience*, vol. 21, no. 11, pp. 1221–1248, 1991.

[115] R. F. Zhu and T. Takaoka, "On improving the average case of the Boyer-Moore string matching algorithm," *Journal of Information Processing*, vol. 10, no. 3, pp. 173–177, 1987.

[116] T. Berry and S. Ravindran, "A fast string matching algorithms and experimental results," in *Proceedings of the Prague Stringology Club Workshop'99*, J. Holub and M. Simánek, Eds., Collaborative Reoprt DC-99-05, Czech Technical University, Prague, Czech Republic, 1999, pp. 16–26.

[117] P. D. Smith, "Experiments with a very fast substring search algorithm," *Software–Practical and Experience*, vol. 21, no. 10, pp. 1065–1074, 1991.

[118] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms.* 5, ANsari Road Darya Ganj, New Delhi-110 002: Galgotia Publications (P) Ltd., 1993, ch. 1, p. 27.

[119] J. Jájá, *An Introduction to Parallel Algorithms.* Reading, MA, USA: Addison-Wesley publishing company, 1992, ch. 7, pp. 311–365.

[120] M. Crochemore, Z. Galil, L. Gasieniec, K. Park, and W. Rytter, "Constant-time randomized parallel string matching," *SIAM Journal on Computing*, vol. 26, no. 4, pp. 950–960, August 1997.

[121] U. Z. T. Goldberg, "Faster parallel string-matching via larger deterministic samples," *Journal of Algorithms*, vol. 16, no. 2, pp. 295–308, March 1994.

[122] Z. Galil, "A constant-time optimal parallel string-matching algorithm," *Journal of the Association for Computing Machinery*, vol. 42, no. 4, pp. 908–918, July 1995.

[123] J. Misra, "Derivation of a parallel string matching algorithm," *Information Processing Letters*, vol. 85, no. 5, pp. 255–260, March 2003.

[124] ——, "Powerlist: A structure for parallel recursion," *ACM transactions on programming languages and systems*, vol. 16, no. 6, pp. 1737–1767, November 1994.

[125] 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.

[126] A. A. Bertossi and F. LOGI, "Parallel string-matching with variable-length don't cares," *Journal of Parallel and Distributed Computing*, vol. 22, no. 2, pp. 229–234, August 1994.

[127] Y. Takefuji, T. Tanaka, and K. C. Lee, "A parallel string search algorithm," *IEEE Transactions on Systems Man and Cybernetics*, vol. 22, no. 2, pp. 332–336, March/April 1992.

[128] 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.

[129] M. E. Isenman and D. E. Shasha, "Performance and architectural issues for string matching," *IEEE Transactions on Computers*, vol. 39, no. 2, pp. 238–250, Feb. 1990.

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

[131] M. J. Foster and H. T. Kung, "The design of special-purpose VLSI chips: Example and opinions," in *Proceedings of the 7th annual symposium on Computer Architecture*, International Conference on Computer Architecture. La Baule, United States: ACM Press, New York, NY, USA, May 1980, pp. 300–307.

[132] A. Mukherjee, "Hardware algorithms for determining similarity between two strings," *IEEE Transactions on Computers*, vol. 38, no. 4, pp. 600–603, Apr. 1989.

[133] 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.

[134] P. D. Michailidis and K. G. Margaritis, "Parallel architecture for flexible approximate text searching," in *CD-ROM Proceedings of the 7th WSEAS International Multiconference on Circuits, Systems, Communications and Computers (WSEAS-CSCC'2003)*, Corfu, Greece, July 2003.

[135] ——, "Bit-level processor array architecture for flexible string matching," in *Proceedings of the 1st Balkan Conference in Informatics (BCI'2003)*, Thessaloniki, Greece, November 2003, pp. 517–526.

[136] K. R. R. Sastry, N. Ranganathan, "CASM – a VLSI chip for approximate string-matching," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 8, pp. 824–830, August 1995.

[137] 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-*

actions on Design Automation of Electronic Systems, vol. 5, no. 3, pp. 682–704, July 2000.

[138] F. Elguibaly, "A fast parallel multiplier–accumulator using the modified booth algorithm," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 902–908, 2000.

[139] ——, "Merged inner-prodcut processor using the modified booth algorithm," *Canadian Journal of Electrical and Computer Engineering*, vol. 25, no. 4, pp. 133–139, 2000.

[140] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*, 2nd ed. Addison-Wesley, 1992.

[141] J. Teich, L. Thiele, and L. Zhang, "Partitioning processor arrays under resource constraints," *The Journal of VLSI Signal Processing*, vol. 17, no. 1, pp. 5–20, Sept. 1997.

[142] K. N. Ganapathy, "Mapping regular recursive algorithms to fine-grained processor arrays," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 1994.

[143] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Transactions on Computers*, vol. 35, no. 1, pp. 1–12, Jan. 1986.

[144] Z. Chen and W. Shang, "Mapping of uniform dependence algorithms onto fixed size processor arrays," in *Proceedings of Seventh International Parallel Processing Symposium*, Apr. 1993, pp. 804–809.

[145] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the Fifteen annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Jan. 1988, pp. 319–329.

[146] W. Shang and J. A. B. Fortes, "On time mapping of uniform dependence algorithms into lower dimensional processor arrays," *IEEE Transaction on Parallel and Distributed Systems*, vol. 3, no. 3, pp. 350–363, May 1992.

[147] J. Xue and P. Lenders, "Avoiding data link and computational conflicts in mapping nested loop algorithms to lower-dimensional processor arrays," in *International Conference on Parallel and Distributed Systems*, Dec. 1994, pp. 567–572.

[148] J. A. Fernando and J. S. N. Jean, "Processor array design with FPGA area constraint," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 18, no. 3, pp. 253–264, Mar. 1999.

[149] S. Siegel and R. Merker, "Algorithm partitioning including optimized data-reuse for processor arrays," in *International Conference on Parallel Computing in Electrical Engineering*, Sept. 2004, pp. 85–90.

[150] H. S. Kim, K. J. Lee, J. Kim, and K. Y. Yoo, "Partitioned systolic multiplier for GF(2^m)," in *International Workshops on Parallel Processing*, Sept. 1999, pp. 192–197.

[151] X. Chen and G. M. Megson, "A general methodology of partitioning and mapping for given regular arrays," *IEEE Transaction on Parallel and Distributed Systems*, vol. 6, no. 10, pp. 1100–1107, Oct. 1995.

[152] P. Tsanakas, N. Koziris, and G. Papakonstantinou, "Chain grouping: a method for partitioning loops onto mesh-connected processor arrays," *IEEE Transaction on Parallel and Distributed Systems*, vol. 11, no. 9, pp. 941–955, Sept. 2000.

[153] H. Yamada, M. Hirata, H. Nagai, and K. Takahashi, "A high-speed string-search engine," *IEEE Journal of Solid-State Circuits*, vol. 22, no. 5, pp. 829–834, Oct. 1987.

[154] M. Hirata, H. Yamada, H. Nagai, and K. Takahashi, "A versatile data string-search VLSI," *IEEE Journal of Solid-State Circuits*, vol. 23, no. 2, pp. 329–335, Apr. 1988.

[155] F. Gebali, *Computer Communication Networks: Analysis and Design*, 3rd ed. Victoria, B.C., Canada: Northstar Digital Design, Inc., 2005.

[156] (2006, Feb.) Maplesoft. html. [Online]. Available: https://webstore.maplesoft.com/

[157] (2005, Nov.) Stratix II devices: The biggest & fastest fpgas. st2-index.jsp. [Online]. Available: http://www.altera.com/products/devices/stratix2/

[158] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, pp. 490–500, 1960.

[159] D. M. Hawken, P. Townsend, and M. F. Webster, "The use of dynamic data-structures in finite-element applications," *International Journal for Numerical Methods in Engineering*, vol. 33, no. 9, pp. 1795–1811, June 1992.

[160] M. Zandieh, "Evaluation of an LC-trie algorithm for IP-address lookups," Master's thesis, Uppsala University, Uppsala, Sweden, 1999, ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T–99-10–SE.pdf.

[161] D. Pao and Y. K. Li, "Enabling incremental updates to LC-trie for efficient management of IP forwarding tables," *IEEE Communications Letters*, vol. 7, no. 5, pp. 245–247, May 2003.

[162] W. S. L. Devroye, "Probabilistic behavior of asymmetric level compressed tries," *Random Structures and Algorithms*, vol. 27, no. 2, pp. 185–200, Sept. 2005.

[163] D. Pao and Y. K. Li, "Enabling incremental updates to LC-trie for efficient management of IP forwarding tables," *IEEE Communications Letters*, vol. 7, no. 5, pp. 245–247, May 2003.

[164] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication.* Cannes, France: ACM Press, New York, NY, USA, 1997, pp. 3–14.

[165] M. A. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8–23, March-April 2001.

[166] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani, "Scalable, memory efficient, high-speed IP lookup algorithms," *IEEE/ACM Transactions on Networking*, vol. 13, no. 4, pp. 802–812, Aug. 2005.

[167] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Transactions on Computer Systems*, vol. 17, no. 1, pp. 1–40, Feb. 1999.

[168] S. Sahni and K. S. Kim, "Efficient construction of multibit tries for ip lookup," *IEEE/ACM Transactions on Networking*, vol. 11, no. 4, pp. 650–662, Aug. 2003.

[169] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *IEEE Proceedings of INFOCOM'98*, vol. 3, San Francisco, CA, 1998, pp. 1240–1247.

[170] Y. A. Reznik, "Some results on tries with adaptive branching," *Theoretical Computer Science*, vol. 289, no. 2, pp. 1009–1026, Oct. 2002.

[171] I. Ioannidis, A. Grama, and M. Atallah, "Adaptive data structures for ip lookups," in *IEEE Proceedings of INFOCOM 2003*, vol. 1, 2003, pp. 75–84.

[172] H. Lu, K. S. Kim, and S. Sahni, "Prefix and interval-partitioned dynamic IP router-tables," *IEEE Transactions on Computers*, vol. 54, no. 5, pp. 545–557, May 2005.

[173] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, and D. B. Parlour, "Scalable IP lookup for internet routers," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 522–534, May 2003.

[174] I. Ioannidis and A. Grama, "Level compressed DAGs for lookup tables," *Computer Networks*, vol. 49, no. 2, pp. 147–160, Oct. 2005.

[175] S. Kaxiras and G. Keramidas, "IPStash: a power-efficient memory architecture for IP-lookup," in *Proceedings of 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003*, Dec. 2003, pp. 361 – 372.

[176] H. Liu, "Routing table compaction in ternary CAM," *IEEE Micro*, vol. 22, no. 1, pp. 58–64, 2002.

[177] M. Sundstrom and L. A. Larzon, "High-performance longest prefix matching supporting high-speed incremental updates and guaranteed compression," in *Proceedings of INFOCOM 2005*, vol. 3, Mar. 2005, pp. 1641–1652.

[178] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 397–409, Apr. 2006.

[179] S. Kasnavi, V. C. Gaudet, P. Berube, and J. N. Amaral, "A hardware-based longest prefix matching scheme for TCAMs," in *Proccedings of IEEE International Symposium on Circuits and Systems*, vol. 4, May 2005, pp. 3339–3342.

[180] M. J. Akhbarizadeh, M. Nourani, D. S. Vijayasarathi, and P. T. Balsara, "PCAM: a ternary CAM optimized for longest prefix matching tasks," in *Proceedings on IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Oct. 2004, pp. 6–11.

[181] M. J. Akhbarizadeh and M. Nourani, "Hardware-based IP routing using partitioned lookup table," *IEEE/ACM Transactions on Networking*, vol. 13, no. 4, pp. 769–781, Aug. 2005.

[182] B. Gamache, Z. Pfeffer, and S. P. Khatri, "A fast ternary CAM design for IP networking applications," in *Proceedings of The 12th International Conference on Computer Communications and Networks*, Oct. 2003, pp. 434–439.

[183] (2005, Aug.) Test routing tables. html. [Online]. Available: http://psp1.iit.cnr.it/~mcsoft/ast/ast_data.html

[184] E. M. Reingold and W. J. Hansen, *Data Structures.* Little, Brown & Company, 1983.

[185] T. N. M. Matsumoto, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[186] M. L. Brundage. (2006, Apr.) Random number generation. html. [Online]. Available: http://www.qbrundage.com/michaelb/pubs/essays/random_number_generation

[187] (2006, May) Rfc 2460. html. [Online]. Available: http://www.faqs.org/rfcs/rfc2460.html

[188] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, Mar. 2001.

[189] O. Owolabi, "Dictionary organizations for efficient similarity retrieval," *Journal of Systems and Software*, vol. 34, no. 2, pp. 127–132, Aug. 1996.

[190] S. Fidanova, "Linear array for spelling correction," *Concurrency-Practice and Experience*, vol. 9, no. 10, pp. 967–973, Oct. 1997.

[191] M. J. S. B. Martins, "Spelling correction for search engine queries," in *Lecture Notes in Computer Science: Advances in Natural Language Processing*, J. L. Vicedo, P. M. Barco, R. M. noz, and M. S. Noeda, Eds., vol. 3230. Spain: Springer Berlin/Heidelberg, Oct. 2004, pp. 372–383.

[192] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, May 1998.

[193] M. Vilares, J. Otero, and J. Vilares, "Robust spelling correction," in *Lecture Notes in Computer Science: Implementation and Application of Automata*, J. Farré, I. Litovsky, and S. Schmitz, Eds., vol. 3845. France: Springer Berlin/Heidelberg, June 2006, pp. 319–328.

[194] M. Vilares, J. Otero, and J. Grana, "Spelling correction on technical documents," in *Lecture Notes in Computer Science: Computer Aided Systems Theory*, A. Q. A. R. M. Díaz, F. Pichler, Ed., vol. 3643. Spain: Springer Berlin/Heidelberg, Feb. 2005, pp. 131–139.

[195] J. Holub, C. Iliopoulos, B. Melichar, and L. Mouchard, "Distributed pattern matching using finite automata," *Journal of Automata, Languages, and Combinatorics*, vol. 6, no. 2, pp. 191–204, 2001.

[196] J. Holab and B. Melichar, "Implementation of nondeterministic finite automata for approximate pattern matching," in *Proceedings of the 3rd International WOrkshop on Implementing Automata*, Rouen, France, 1998, pp. 74–81.

[197] S. Mihov and K. U. Schulz, "Fast approximate search in large dictionaries," *Computational Linguistics*, vol. 30, no. 4, pp. 451–477, 2004.

[198] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Reading, MA, USA: Addison-Wesley, Nov. 2000.

[199] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Mountain View, CA, USA: Prentice-Hall, 1999.

[200] D. P. de Farias, "On constraint sampling in the linear programming approach to approximate dynamic programming," *Mathematics of Operations Research*, vol. 29, no. 3, pp. 462–478, Aug. 2004.

[201] D. P. de Farias and B. V. Roy, "The linear programming approach to approximate dynamic programming," *Operations Research*, vol. 51, no. 6, pp. 850–865, Nov. 2003.

[202] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Sov. Phys. Dokl.*, vol. 10, pp. 707–710, Feb. 1966.

[203] (2005, Mar.) Sophos report reveals words that spammers most commonly try to disguise. html. [Online]. Available: http://www.sophos.com/pressoffice/news/articles/2005/03/sa_spamwords.ht%ml

[204] I. Androutsopoulos. (2006) Publications. lingspam_public.tar.gz. [Online]. Available: http://www.aueb.gr/users/ion/data/

[205] (2006, May) Kevin's word list page. html. [Online]. Available: http://wordlist.sourceforge.net/