High Performance Transducer Architecture for Content Inspection Engines

by

Mohammadreza Yazdani

B.Sc., Isfahan University of Technology, 1998

M.Sc., Isfahan University of Technology, 2001

M.A.Sc., Carleton University, 2003

A thesis submitted to the faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Ottawa-Carleton Institute for Electrical and Computer Engineering
Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario

February 2008

Your file Votre référence
ISBN: 978-0-494-40545-1
Our file Notre référence
ISBN: 978-0-494-40545-1

Abstract

This doctoral thesis investigates different aspects of providing a transducer-based solution for the general problem of content inspection. It introduces an efficient architecture, called Two-Level Transducer (TLT) architecture, for implementation of high performance transducers in hardware. The TLT architecture benefits from a two-level memory structure to avoid the memory explosion problem associated with the normal implementation of high-performance transducers. Dynamic programming techniques are developed to optimize the amount of memory required for the TLT implementation of high performance transducers without compromising throughput. We propose to use *worst-case throughput* as an appropriate criterion for speed evaluation of Content Inspection Engines (CIEs) and provide a model for CIEs, which can be used to compute this criterion.

We present Concatenation Transducers (CTs) for solving content inspection problems that need to process packets with contents specified by a non-regular language, such as XML. We also develop an algorithm for calculating the worst-case throughput of CTs and prove its correctness and low polynomial time complexity.

The thesis also investigates the problem of representing range fields in Ternary Content Addressable Memory (TCAM). For a class of minimum-width binary encoding schemes, which includes lexicographic encoding (i.e., standard binary encoding) and binary reflected Gray encoding, we provide a tight lower bound on the worst-case expansion rate of representing ranges in TCAM. Besides, we present two linear-time algorithms for computing a minimum-size TCAM representation of a given range under the lexicographic and Gray encodings.

Acknowledgements

It is my pleasure to thank the many people who made this thesis possible.

First and foremost, I would like to express my profound and sincere gratitude to my main supervisor, Dr. Wojciech Fraczak, for his kind support, enthusiasm, inspiration, and trenchant criticism. Throughput my research at IDT Canada, he provided me encouragement, sound advice, good company, and lots of good ideas. I cannot thank him enough. Simply put: Wojciech rocks!

I am also deeply indebted to my co-supervisor, Prof. Ioannis Lambadaris, for all the stimulating guidance and moral support he offered me throughput the period of this research.

I would like to thank the members of the examining board of my thesis, Prof. Marek Zaremba of Université du Québec en Outaouais, Prof. Jiying Zhao of University of Ottawa, Prof. Michel Barbeau of the school of Computer Science, and Prof. Ashraf Matrawy of the department of Systems and Computer Engineering, for their time and helpful comments.

The three years of my internship in IDT Canada was an excellent opportunity for me to broaden my perspective on the practical aspects in the industry. I would like to thank Feliks Welfeld for hiring me in the research group of IDT Canada, and for his patience and thoughtfulness in answering my technical questions. I also wish to give my sincere thanks to Antoine Fink for his generous friendship and help. My gratitude go to all employees and contractors of IDT Canada, specially Maria Fronczak, Mario

Beaulieu, Cristian Lambiri, Dahir Osman, Aroosh Elahi, Prof. Jurek Czyzowicz, and Prof. Andrej Pelc.

My warm thanks are due to all my friends specially Arash Shokrani, Pirouz Zarrinkhat, Kayvan Mosharraf, Saied Hemati, Maryam Sabbaghian, and Hamid Saeedi for their care and support. I am also grateful to the office administration and technical staff in the Systems and Computer Engineering department of Carleton University, for helping the department to run smoothly and assisting me in many different ways. Darlene Hebert, Anna Lee, Blazenka Power, Coleen Kornelsen, Jennifer Poll, and Cheryl Auclair, deserve special mention.

Last but not least, I owe my loving thanks to my family whose unflagging support and love has enabled me to complete this Ph. D. thesis.

To all those who speak truth to power and those who are a voice for the voiceless

Table of Contents

List of Figures

List of Notations

Notation	Description	First Use
$\emptyset$	The empty language	13
ϵ	The empty string	13
$L_1 \cup L_2$	The union of two languages L_1 and L_2	14
$L_1 L_2$	The concatentation of two languages L_1 and L_2	14
L^*	The Kleene closure of the language L	14
Q	The set of states of a finite state machine/transducer	15, 20
Σ	The set of input symbols of a finite state machine/transducer/concatenation transducer	15, 20, 74
δ	The transition function of a a finite state machine/transducer	15, 20
q_0	The initial state of a finite state machine/transducer	15, 20
F	The set of final states of a finite state machine/transducer	15, 20
$L(A)$	The language of A	17
$\boldsymbol{\emptyset}$	The regular expression denoting the language $\emptyset$	19
Γ	The set of output symbols of a finite state transducer	20
$\wedge$	AND operation	35
$\vee$	OR operation	35
$\neg$	NOT operation	35
B	Symbol-size bound	45

Notations	Description	First Use
$N_{TCAM}(u)$	The number of rules in the TCAM representation of the layout entry corresponding to u	45
$c(u)$	The cost of the node u	45
$c(G)$	The cost of the graph G	46
T_v	The subtree of T rooted at v	47
T'_v	A minimum-cost tree equivalent to T_v	47
$S_v(k)$	The set of all k-descendants of v	47
c_v^d	The cost of a guaranteed surplus d in node v	49
D_v	The set of all surpluses in v	49
$G_{S_v(k)}$	The subgraph of G induced by $S_v(k)$ and all descendants of elements of $S_v(k)$	53
n_G	The number of nodes in the graph G	53
pc_v^d	The pseudo-cost of having surplus d in node v	53
G_v^d	The graph guaranteeing a surplus d at node v with the minimum pc_v^d	53
$w(p)$	the weight of the path p	63
$t(p)$	the transition time of the path p	63
Ω	The set of output symbols of a concatenation transducer	74
S	The set of switch nodes of a concatenation transducer	74
C	The set of concatentation nodes of a concatenation transducer	74
A	The set of accepting nodes of a concatenation transducer	74
η_S	The partial mapping of the switch nodes of a concatenation transducer	74
η_C	The partial mapping of the concatenation nodes of a concatenation transducer	74

Notations	Description	First Use
v_{in}	The initial node of a concatenation transducer	74
$w(e)$	The weight of an edge e	80
$wct(M)$	The worst-case throughput of a concatenation transducer M	80
w_μ	μ-updated weight function	82
(π_u, ϕ)	A c-path with starting node u	85
$\|(\pi_u, \phi)\|$	The length of the c-path (π_u, ϕ)	85
$w(\pi_u, \phi)$	The weight of the c-path (π_u, ϕ)	85
$throughput(\pi_u, \phi)$	The throughput of the c-path (π_u, ϕ)	85
$h(\pi_u, \phi)$	The height of the c-path (π_u, ϕ)	86
$\mathcal{R}$	A TCAM	94
$E(f)$	The maximum expansion rate of an encoding scheme f	97
T_n	An n-bit dense-tree encoding	104
C_L	A left chain	104
C_R	A right chain	104
$\overline{a}$	The complement of a	104
$\texttt{Lex}_n$	The n-bit lexicographic encoding	110
$\texttt{Gray}_n$	The n-bit Gray encoding	110
$\overline{\texttt{Gray}_n}$	The mirror copy of $\texttt{Gray}_n$	116
$\mathcal{S}$	The skeleton tree of a range	117

List of Abbreviations

Abbreviation	Description	First Use
ACL	Access Control List	126
ASIC	Application-Specific Integrated Circuits	5
CIE	Content Inspection Engine	4
CT	Concatenation Transducer	73
DNF	Disjunctive Normal Form	35
DoS	Denial of Service	2
FPGA	Field Programmable Gate Array	5
FSM	Finite State Machine	7
FST	Finite State Transducer	20
HTTP	Hypertext Transfer Protocol	2
IDT	Integrated Device Technology	22
IP	Internet Protocol	1
LCA	Lowest Common Ancestor	117
MWTR	Minimum Weight to Time Ratio	24
NIDS	Network Intrusion Detection Systems	2
PDL	Packet Description Language	22
QoS	Quality of Service	1
TCAM	Ternary Content Addressable Memory	5
TCP	Transmission Control Protocol	64
TLT	Two-Level Transducer	9
XML	Extensible Markup Language	73

Chapter 1

Introduction

In the early days of the Internet, packet processing was confined to forwarding Internet Protocol (IP) packets from one network interface to another. For this purpose, the routers had to make the forwarding decisions based only on the destination address field of the packet headers. As organizations started to use the Internet for their critical functions and transactions, the need for equipping the Internet with Quality of Service (QoS) features and security guarantees against malicious intruders arose. These requirements were initially addressed by introducing multifield packet classification processing into some network devices, including routers and firewalls [1–4]. Multifield packet classification involves processing a number of packet headers which contain information such as source address, destination address, and packet type (e-mail, web-traffic, etc.).

Incorporation of multifield packet classification into routers enabled them to provide limited QoS and security services such as control over resources (bandwidth, equipment), tailored services, and protection against simple connection based attacks. However, such limited inspection of packets cannot, for example, distinguish packets containing valid data from those containing harmful data, if they originate from an otherwise trustful source such as an ISP's e-mail server. The growing needs

for content-aware services and network security against threats hidden deep in application headers and payload, can be met through processing the whole contents of the network flow packets. This processing, which involves both the headers and the contents of IP packets is referred to as *Content Inspection.*

The trend of shifting the boundaries of packet processing from packet headers to its contents is visible in several modern packet processing applications. For instance, signature-based Network Intrusion Detection Systems (NIDS) [5] match the contents of packets against a set of predefined signature strings to identify and deter malicious intrusions or Denial of Service (DoS) attacks [6] over the network. Content-based billing systems [7] are another example of emerging packet processing applications in which media packets are analyzed in order to bill the receiver based on material transferred over the network. Likewise, in load-balancing systems [8], content forwarding switches look at the HTTP headers and distribute the request among predetermined servers.

We continue this chapter by a formal definition of the Content Inspection Problem. Then we compare software-based solutions against hardware-based solutions and also refer to the major challenges associated with the development of a solution to this problem.

1.1 Content Inspection Problem: Definition, Challenges and Solutions

The content inspection problem identifies the flow that a packet belongs to, based on the header and the payload of the packet. This identification is normally performed based on a collection of *prioritized policy rules.* Each rule specifies a *class* that a packet may belong to. There is an identifier (Class ID) associated to each class which uniquely determines the *action* associated with the rule. The content inspection process consists of repeatedly reading an incoming packet, finding the highest priority

rule that matches the read packet, and taking the action that is associated with the matched rule. Note that this general definition covers most of the network content-aware services and security functions. Since the definition of rules is not limited to a specific part of the packet, a rule can be defined on the header or payload of the packet and even on a stream of bits obtained from concatenating the contents of a number of packets. For instance, a rule can be defined to filter out all the packets that include the word "foo", anywhere in their contents, with the associated action "block the packet".

1.1.1 Software-Based and Hardware-Based Solutions

Traditional implementations of packet processing devices such as firewalls are software-based solutions designed around CPU cores, where data packets are read from memory into CPU registers and then the CPU performs certain processing on data before queuing processed packets for transmission [9–12]. For such solutions, the speed of processing is determined by CPU clock, number of CPU cores working in parallel, memory access time, and also number of times that the processing algorithm needs to read or copy a single packet from or into memory. At data rates of several giga bits per second, where every packet should be processed in a few microseconds at most, such a software-based packet processing architecture cannot be adopted for performing the huge amount of processing required for content inspection [13].

Today, there are more specialized hardware-based approaches to this problem which allow for a much higher level of packet processing efficiency. The main idea behind these hardware-based architectures is to offload the packet payload processing required for content inspection, from the CPU of the network processing unit. For this purpose, a standalone content inspection engine is used for content inspection that places information about the packet or flow contents into the packet header. Then the CPU only reads this contents-inspection result and performs the appropriate filtering

operation on each packet.

In the rest of this thesis, we use the term Content Inspection Engine (CIE) to generally refer to a solution provided for the content inspection problem, independent from its structure (i.e., it can be a software-based or a hardware-based solution).

1.1.2 The Challenges Associated with Content Inspection and Filtering

Content inspection processing, i.e., performing packet processing up to the application layer, is more difficult than the packet processing at the lower layers. As such designing a CIE is a complex task associated with several design challenges and trade-offs. The most significant of these challenges are the following:

- Providing wire-speed processing.
- Fitting inspection rules into available memory.
- Supporting the dynamic nature of inspection policy.

Providing Wire-Speed Processing

An implicit requirement of content inspection is that incoming packet stream should be inspected at wire speed. This means that the CIE should process the incoming bit stream at a speed not smaller than the incoming bit rate. This is an essential consideration in the design process of any CIE as otherwise, if the incoming line-speed is greater than processing speed, the system would require an unbounded amount of memory for buffering the unprocessed packets built up on the input side. This requirement together with the line rates as fast as several giga-bits per second, and complex application layer processing of content inspection, creates a unique processing challenge for the designer.

Fitting Inspection Policy into Available Memory

Inspection policy is the set of rules, each rule consisting of some patterns and signatures, which is used by a CIE for monitoring network traffic. The CIE compares the stream of network packets with the patterns of inspection policy rules to manage network resources, detect intrusions, or monitor viruses, worms, and Trojans. Regardless of the architecture used for the processing core of the CIE, the inspection policy must be stored somewhere.

There are various types of memory that can be used for storing inspection policy, including dynamic RAM, static RAM, and Ternary Content Addressable Memory (TCAM) [14].

Supporting the Dynamic Nature of Inspection Policy

The inspection policy of CIE needs to be updated with new rules and patterns that, for example, enables it to detect and prevent new threats such as fast-spreading viruses and worms. To address this requirement, a real-time automatic updating infrastructure should be designed to update the policy with the latest changes as they emerge.

1.2 Related Work and Our Motivations

Content inspection being a serious performance bottleneck of the Internet, it has received a significant attention in last few years, specially from the industrial community [15–19]. Existing hardware based solutions such as [16, 18], that use custom Application-Specific Integrated Circuits (ASICs) require costly, time-consuming development resources and offer little flexibility to adopt new rules. Field-Programmable Gate Array (FPGA) based solutions benefit from the reconfigurable hardware capability and parallelism of FPGAs. For example, the specialized hardware of [20] is

based on the fact that each content inspecting rule represents an independent pattern matching process. So it uses a parallel architecture to compare each packet against all the rules in parallel. While these techniques provide a better programmability than the custom ASIC solutions, their resource requirements are proportional to the total number of rules. Therefore, scalability with the policy rule size is the primary concern with such solutions.

There has also been more recent work on hardware-implementable multiple-pattern search algorithms [21–23]. Motivated by the fact that pattern matching dominates the performance of many content inspection systems, these works concentrate their efforts on building smaller and faster pattern matching algorithms. The deterministic pattern matching hardware-based algorithm of [23] applies bit-map compression to the Aho-Corasick state machine to gain both compact storage and worst-case performance of two memory accesses per 8-bit state transitions. The algorithm assumes the availability of very large memory widths which are implementable only in modern ASIC. Although this algorithm has a deterministic worst-case lookup time, it matches only a single 8-bit character per state transition.

The multiple pattern matching solution of [21] makes use of parallel Bloom filters [24]. This approach builds an on-chip Bloom filter for each pattern length. The Bloom filters work as an imprecise pattern matching algorithm for the payload strings. Upon a positive Bloom filter match, a precise pattern matching algorithm is used to verify the presence of pattern. The algorithm limits the feasible pattern length to a maximum value and thus cannot be used for applications where pattern lengths vary from tens to thousands of bytes. Besides, the worst-case performance of this algorithm could be unsatisfactory because a handcrafted stream of packets may overwhelm the exact matching algorithm by introducing a high rate of sequential false positives.

A TCAM based solution for the multiple pattern matching problem is proposed in [22]. In this algorithm, the long patterns are cut into smaller patterns of width w and

stored in TCAM. The algorithm starts with looking up the first w bytes of the packet payload into TCAM and shifts one byte at a time. At each position, if it matches a TCAM entry, the result is recorded in a hit list and also the hit list is consulted to verify if a complete pattern has matched. The throughput of this algorithm is limited to a single 8-bit character per two memory accesses and an average performance between 1 and 2 memory access per character is reported.

Most of the existing algorithmic solutions for the general problem of content inspection focus on developing a dedicated hardware for a specific content inspection processing (for example, multiple pattern matching). Besides, these solutions are usually concerned only with the average-case performance of the systems and does not provide guarantees for the worst-case speed performance. In this thesis, we have focused on the development of a more general solution that could be applied to a wider range of content inspection problems. We have also been concerned with the worst-case speed performance of our solution from the very beginning. These motivations have led us to develop a solution over the concept of Finite State Machines (FSMs). Using a state machine for solving the problem provides us with several advantages which we refer to some of them here. We will benefit from these advantages to obtain the worst-case processing speed of a CIE and also calculate the amount of memory required for every content inspection problem in next chapters:

- The execution time required for solving a problem by an FSM is predictable.

- The amount of memory required for implementing an FSM is directly proportional to the number of states in the FSM.

While the main motivation of the thesis is to provide an efficient solution for the content inspection problem, it can also be considered as a research work towards resolving the memory issues related to the hardware architecture of large and high-performance FSMs, independent from the application of the state machine.

1.3 Our Approach and Contributions

The solution presented in this thesis for the content inspection problem is based on FSMs. In fact, one can observe that the content inspection problem can be represented and solved by an FSM. In order to clarify this observation we first give an informal introduction to FSMs (the formal definitions of the concepts related to FSMs will be provided in the next chapter) and then explain how a very simple and abstract content inspection problem can be naturally solved by an FSM.

An FSM is a useful model for many software or hardware systems which can be considered, at any time, to be in one of a finite number of "states". A state remembers the relevant portion of system's history, i.e., it reflects the input changes from the start time of the system to present. The FSM starts working from its initial state and makes a state transition based on the value of its first input symbol. It then continues to make state transitions based on the values of the next input symbols until it enters to a final state where it performs the "action" associated with that final state.

Let us consider the following content inspection problem: we want to look at the first 3 bits of each incoming packet (e.g., the first 3 bits of the source address in IP packets) and classify the packets into two classes A and B, if their first three bits are or are not equal to 000, respectively. An obvious solution to this problem would initially look at the first bit of the packet. If it is 0 then it proceeds to the second bit and if the second bit is 0 either, it checks the third bit to make the right classification decision. If any of the first two bits are 1, however, the next bits can be skipped and the packet could be classified as a packet of class B immediately. We can illustrate the rules of this 3-stage packet processing procedure by the diagram of Figure 1.1.

This diagram is in fact a representation of an FSM which works in a number of steps as follows. At each step of the processing, the state machine is in one of the states which are represented by circles in the diagram. This state remembers the

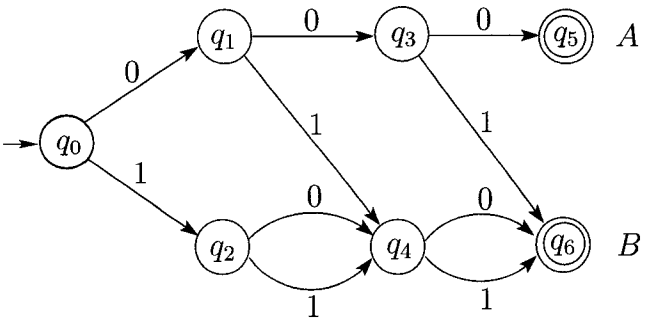


Figure 1.1: Illustration of a packet processing procedure.

results of the processing that the machine has done on the read bits of the packet so far (at the beginning we have only one circle because we have not read any bit of the packet yet). In each state, the state machine reads the first unread bit of the incoming packet and changes its state accordingly. For example, if after processing the first two bits of the packet, the state machine is in state q_3 and the third bit of the packet is zero, it changes its state to q_5 and classifies the packet as a packet of class A.

1.3.1 Contributions

The contributions of this thesis are summarized as follows:

- Proposing to design CIEs as *high-performance transducers*; this technique enables us to provide wire-speed processing for CIEs (Chapter 3).

- Introducing the Two-Level Transducer (TLT) architecture as an efficient architecture for hardware implementation of high-performance transducers. This architecture uses a two-level memory structure to avoid the memory explosion problem associated with the existing methods of implementing high-performance transducers (Chapter 3).

- Development of a procedure, based on dynamic programming techniques, for

optimizing the memory requirements of implementing high-performance transducers by the TLT architecture (Chapter 3).

- Proposing the *worst-case throughput* as a new architecture-independent criterion for speed evaluation of CIEs (Chapter 3).
- Development of a method, based on a graphical model of CIEs, for calculating the worst-case throughput of CIEs (Chapter 3).
- Validating the efficiency of the TLT architecture through performance evaluation of a real-life CIE based on this architecture (Chapter 3).
- Introducing a new algorithm for calculating the worst-case throughput of Concatenation Transducers and proving it correctness (Chapter 4).
- Presenting a systematic approach for investigating the problem of range representation in TCAM by providing a formal definition of the problem (Chapter 5).
- Providing a tight lower bound on the worst-case expansion rate of range representation in TCAM for a family of minimum-width encoding schemes (Chapter 5).
- Introducing two linear-time algorithms for calculating a minimum-size TCAM representation of a given range under the lexicographic and the binary reflected Gray encodings (Chapter 5).

Some parts of the results of this thesis have been presented and published in a number of conferences and a journal [25–29].

1.4 Organization of the Thesis

Chapter 2 provides a review of the basic concepts of FSMs and transducers. The concepts and definitions of this chapter will be used through the rest of the thesis.

The contributions of the thesis are presented in Chapters 3, 4, and 5.

In Chapter 3, we introduce our architecture for efficient hardware implementation of high-speed CIEs. Besides, we present dynamic programming techniques for minimizing the memory requirements of this architecture. We also introduce *worst-case throughput* as an appropriate criterion for speed evaluation of CIEs. At the end of this chapter we will present some simulation results to show the efficiency of the TLT architecture.

In Chapter 4 we extend the set of content inspection problems which can be solved in hardware by our proposed architecture. We present CTs as a solution to the content inspection problems which process packets with contents described by simple languages. We also present an algorithm for calculating the worst-case throughput of CTs and prove its correctness.

In Chapter 5, we explore the problem of range-representation in TCAM. We start by providing a formal definition for this problem. Then, for a class of binary encoding schemes, we provide a tight lower bound on the worst-case expansion rate of the range representation in TCAM. Finally, as the main contribution of this chapter, we present two linear-time algorithms for generating the minimal TCAM representation of a given range for the lexicographic and Gray encoding.

Chapter 6 briefly summarizes the contributions of the thesis, and suggests some directions for future research work.

Chapter 2

Preliminaries and Background

This chapter serves as a summary of the basic concepts related to FSMs that will be involved in the rest of the thesis. The study of state machines and their applications has been an important part of the core of Computer Science for several decades. Consequently it has a rich and varied literature which is covered in many of the today's available books on this area (for example [30–33]). We focus on terms and definitions that will be used in the explanation of our state machine based solutions for the problem of content inspection.

2.1 The Central Concepts

This section presents some of the concepts and definitions that will be used for introducing FSMs and their properties.

2.1.1 Alphabet

An *alphabet* is a finite and nonempty set of symbols. In our case, the most common alphabet is $\{0, 1\}$, i.e., the binary alphabet. Other common alphabets include $\{a, b, ..., z\}$ (i.e., the set of all lowercase letters) and the set of all ASCII characters.

2.1.2 Strings

A *string* is a finite sequence of symbols from an alphabet. For example, 10011 and 000 are two binary strings drawn from the binary alphabet $\{0.1\}$. The *length* of a string w is defined as the number of alphabet symbols in w and is denoted by $|w|$. For example $|1001001| = 7$ and $|1100| = 4$.

The Empty String

The *empty string* is the string of length zero. This unique string, denoted as ϵ, may be chosen from any arbitrary alphabet.

Concatenation of Strings

For any two strings x and y, the *concatenation* of x and y, denoted by xy, is defined as the string formed by joining y to the end of x. For example, if $x = 10011$ and $y = 000$, then $xy = 10011000$ and $yx = 00010011$. Note that for any string w, we have $w\epsilon = \epsilon w = w$.

2.1.3 Languages

A *language* L over an alphabet Σ is defined as a set of strings over Σ. While strings of a language are drawn from a fixed and finite alphabet, the language can have an infinite number of strings. The empty language is a language over any alphabet and is denoted by $\emptyset$.

Rational Operations on Languages

Language operations are used to generate new languages from given languages. For two languages L_1 and L_2 over some common alphabet, we define the following language operators:

1- The *union* of L_1 and L_2, denoted by $L_1 \cup L_2$, is the set of all strings which are contained in either L_1 or L_2, or both.

2- The *concatenation* of L_1 and L_2, denoted by L_1L_2, is the set of all strings of the form vw where v is a string from L_1 and w is a string from L_2.

3- the *Kleene closure* of the language L, denoted by L^*, is the set of all strings that can be formed by taking any number of strings from L, possibly with repetitions, and concatenating all of them. For example, the set of all strings over alphabet Σ is Σ^*.

Example 2.1.1. *For two languages* $L_1 = \{01, 111, 10011\}$ *and* $L_2 = \{\epsilon, 111\}$ *we have:*

$$L_1 \cup L_2 = \{\epsilon, 01, 111, 10011\}$$

$$L_1L_2 = \{01, 111, 10011, 01111, 111111, 10011111\}$$

$$L_2^* = \{\epsilon, 111, 111111, 111111111, ...\}$$

◀

2.2 Finite State Machines

In the informal introduction of FSMs in Chapter 1, we restricted the definition of an FSM to a system that is in a single state after reading any sequence of input symbols. Such an FSM is referred to as a "deterministic" FSM, where the term deterministic refers to the fact that for each current state there is at most one transition for each possible input. In contrast, in "non-deterministic" FSMs, there can be none, one, or more than one transition from a current state for a given possible input. In

this section we formally define deterministic FSMs and refer to the algorithms for minimizing them. As we will not deal with non-deterministic FSMs, we maintain the convention that all our FSMs are deterministic.

2.2.1 Finite State Machines

An FSM is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite nonempty set of *states.*

- Σ is a finite non-empty set of *input symbols* (also called the *alphabet*).

- δ is a partial mapping called *transition function*:

$$\delta : Q \times \Sigma \rightarrow Q,$$

i.e., for a state q and an input symbol a, the state transition function returns a single state $\delta(q, a)$.

- $q_0 \in Q$ is the *initial state.*

- $F \subseteq Q$ is the set of *final* states (also called *accepting* states).

Example 2.2.1. *Let us consider an FSM* $A = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\})$, *with the transition function* δ *defined as:*

$$\delta(q_0, a) = q_0, \delta(q_0, b) = q_1$$

$$\delta(q_1, a) = q_2, \delta(q_1, b) = q_1$$

$$\delta(q_2, a) = q_2, \delta(q_2, b) = q_2$$

For this FSM, the set of states is $\{q_0, q_1, q_2\}$, *the set of input symbols is* $\{a, b\}$, *and the initial and accepting states are* q_0 *and* q_2, *respectively.* ◀

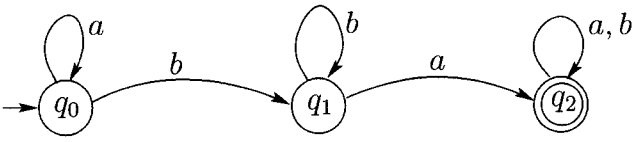


Figure 2.1: The state diagram for the FSM of Example 2.2.1.

State Diagram Representation of an FSM

It is convenient to represent an FSM by a graph called *state diagram.* A state diagram for an FSM $A = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

- Each state in Q is represented by a node (i.e., a small circle).

- Let $\delta(q, a) = q'$, for two states q and q' in Q and an input symbol a in Σ. Then there is an arc pointing from node q to node q', labeled with symbol a. If there are more than one input symbols that cause transitions from state q to state q', they are represented by a single arc, labeled with the list of symbols.

- The initial state is distinguished by a short arrow pointing at its corresponding node from nowhere.

- Nodes corresponding to states in F (accepting states) and those not in F are designated by double circles and single circles, respectively.

Example 2.2.2. *Figure 2.1 shows the state diagram for the FSM A that was defined in Example 2.2.1. The three nodes of this diagram correspond to the three states of A. Also the initial state q_0 and the final state q_2 are distinguished by an incoming arrow and a double circle, respectively. The arcs of this state diagram reflect the transition function of Example 2.2.1.* ◀

String Processing

An FSM is a machine for processing sequences of *input strings.* The input strings are read from the so-called input *tape* of an FSM $A = (Q, \Sigma, \delta, q_0, F)$ and processed

as follows. Beginning from its initial state q_0, A starts processing the input string $w = a_1a_2 \ldots a_n$ by consulting the transition function δ to find the state $q_1 = \delta(q_0, a_1)$ that it should enter after reading the first input symbol a_1 from the input tape. After going to state q_1, it reads the next input symbol a_2 and finds its next state $q_2 = \delta(q_1, a_2)$. The process continues by successively reading the next input symbols $a_3a_4 \ldots a_n$ from the input tape and changing the current state to $q_3q_4 \ldots q_n$ such that $\delta(q_{i-1}, a_i) = q_i$ for each i. If q_n happens to be an accepting state, we say that A *accepts* the input string w. Otherwise, we say that A *rejects* w. For example the FSM of Examples 2.2.1 and 2.2.2 accepts the string *aabab* but rejects the strings ϵ, b and *aabb*.

The Language and the Equivalence of FSMs

The language of an FSM A is defined as the set of all strings that are accepted by A and is denoted by $L(A)$. If a language L is $L(A)$ for some FSM A, then we call L a *regular language*. We say that two FSMs are *equivalent* if they define the same language.

Example 2.2.3. *The language accepted by the FSM A of Examples 2.2.1 and 2.2.2 is the set of all strings of a's and b's which have the sequence 'ba' as a factor, i.e., the substring 'ba' occurs somewhere in the string.* ◀

Minimization of FSMs

An FSM can be implemented in hardware by programmable logic devices. Such a hardware implementation requires a register to store the current state of the FSM, a block of combinatorial logic to determine the state transition, and some amount of memory to store the states (together with their transitions). The amount of required memory is directly related to the number of states of the FSM. As such, the problem

of minimizing the amount of hardware resources required for implementing an FSM translates to the following problem:

For an FSM A, how we can find an equivalent FSM that has as few states as any FSM accepting the same language $L(A)$?

Among the algorithms of finding the minimum-state FSM, the algorithm presented by Hopcroft in [34], which runs in $n \log n$ (n is the number of states of the FSM), is the most efficient algorithm.

2.3 Regular Expressions

Regular expressions are an algebraic language-definition notation. They are usually used to give a concise description of a language without listing all strings of the language. It has been shown in [35] that regular expressions can define the same languages that FSMs describe: the regular languages. Regular expressions also provide us with a convenient way to express the strings that we want to accept. As a consequence regular expressions are used as the input language for many systems that process strings, including: lexical analyzer generators such as Lex [36], and also many text editors and utilities such as Grep [37] and Perl [38] that search and modify bodies of text based on certain patterns.

2.3.1 Building Regular Expressions

Regular expressions consist of constants and variants that denote languages, and operators for the operations over these languages. Here we present a recursive description of regular expressions, which for each regular expression E, describes the language $L(E)$ that it represents:

For a finite alphabet Σ, the following constants and operations are defined.

Constants:

1. $\boldsymbol{\emptyset}$ is a regular expression, denoting the language $\emptyset$. That is $L(\boldsymbol{\emptyset})$ is the empty language.

2. $\boldsymbol{\epsilon}$ is a regular expression, denoting the language $\{\epsilon\}$. That is $L(\boldsymbol{\epsilon}) = \epsilon$.

3. For every a $\in \Sigma$, **a** is a regular expression denoting the language {a}. That is $L(\mathbf{a}) = \{\text{a}\}$.

Operations:

1- If R_1 and R_2 are two regular expressions, then $R_1 + R_2$ is a regular expression denoting the union of $L(R_1)$ and $L(R_1)$. That is $L(R_1 + R_2) = L(R_1) \cup L(R_2)$.

2- If R_1 and R_2 are two regular expressions, then R_1R_2 is a regular expression denoting the concatenation of $L(R_1)$ and $L(R_2)$. That is $L(R_1R_2) = L(R_1)L(R_2)$.

3- For any regular expression R, R^* is a regular expression denoting the Kleene star closure of $L(R)$. That is $L(R^*) = (L(R))^*$.

For precedence we use ordinary parentheses, (). Also notice that $L(\boldsymbol{\emptyset}^*) = \{\epsilon\}$.

Example 2.3.1. *Let us derive a regular expression for the language described by the FSM of Figure 2.1; the set of all strings of a's and b's which have the sequence 'ba' somewhere in the string. We first build the regular expression for the language $\{ba\}$ by concatenating the expressions denoting the languages $\{b\}$ and $\{a\}$; this expression is* $\boldsymbol{ba}$. *Then we represent the set of all strings on a and b by* $(\boldsymbol{a}+\boldsymbol{b})^*$. *Now the regular expression for the language in question would be:*

$$(\boldsymbol{a}+\boldsymbol{b})^*\boldsymbol{ba}(\boldsymbol{a}+\boldsymbol{b})^*$$

◀

2.4 Finite State Transducers

An FSM processes input strings read from its input tape, by computing a function that maps strings into the set $\{accept, reject\}$. A Finite State Transducer (FST) is an FSM with two tapes: one input tape and one output tape. Each input stings of the input tape is processed by the FST and a corresponding output string is put on the output tape. We do not consider nondeterministic FSTs where more than one output string may be generated for an input string.

2.4.1 Finite State Transducers

An FST is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ such that:

- Q is a finite nonempty set of *states.*
- Σ is a finite non-empty set of input symbols, called the *input alphabet.*
- Γ is a finite non-empty set of output symbols, called the *output alphabet.*
- δ is a partial mapping called *transition function*:

$$\delta : Q \times \Sigma \rightarrow Q \times \Gamma^*$$

In other words, for a state q and an input symbol a, the state transition function returns a single state q' and an output string $w \in \Gamma^*$.

- $q_0 \in Q$ is the *initial state.*
- $F \subset Q$ is the set of *final* or *accepting* states.

State Diagram Representation of an FST

State diagram representation of FSTs has the same structure of state diagrams for FSMs except for the state transitions which are labeled with one symbol and one string, instead of one symbol. The symbol, chosen from the input alphabet, represents

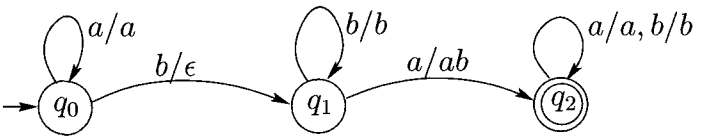


Figure 2.2: The state diagram of an FST.

the input symbol and the string, chosen from the language of the output alphabet, represents the output string corresponding to that transition.

Example 2.4.1. *Consider an FST* $A = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b\}, \delta, q_0, \{q_2\})$, *with the transition function* δ *defined as:*

$$\delta(q_0, a) = (q_0, a), \delta(q_0, b) = (q_1, \epsilon)$$

$$\delta(q_1, a) = (q_2, ab), \delta(q_1, b) = (q_1, b)$$

$$\delta(q_2, a) = (q_2, a), \delta(q_2, b) = (q_2, b)$$

The state diagram of this FST is shown if Figure 2.2. This FST accepts all the strings which contain a substring 'ba' and generates an output string which is the same as the input string except for the first substring 'ba' which is replaced by the substring 'ab'. Note that if an input string does not contain the substring 'ba', the transducer produces some output. However, the string is not accepted because the transducer does not go to its final state. ◀

2.4.2 Packet Processing Using FSTs

Transducers are useful devices for computing functions on the strings (i.e., translating one string to another). In this thesis, we will use transducers for packet processing. In this application, the input string of the transducer is the packet contents and the output string is the result of processing. This means that the input alphabet is always

$\{0, 1\}$. The output alphabet, however, depends on the outputs that are expected to be produced as the result of processing and may change from one content inspection problem to another. For example, if the purpose of processing is to either "Accept" or "Reject" the packet, without generating any other output sting during processing of the packets, then the output alphabet would be $\{Accept, Reject\}$.

The transducer which will be used for a specific packet processing application is specified by the goals of that packet processing application. The goals of the processing are expressed by a set of so called *policy rules*, where a set of policy rules may be expressed by different means including regular expressions and specialized packet processing languages such as the Packet Description Language (PDL) of Integrated Device Technology (IDT) [39, 40].

Chapter 3

High-Performance Finite State Transducers for Network Packet Processing

In this chapter, we focus on development of an efficient mechanism for implementing an FST in hardware, which allows for a high processing speed. This mechanism overcomes the "one-bit per state-transition" speed limit of an FST by transforming it to a *high-performance* FST. A high-performance FST can process several bits in each of its states to provide a wire-speed inspection of the flow of incoming data, according to the set of inspection policy rules.

We introduce a specialized hardware-based architecture for implementing high-performance FSTs which has a high level of efficiency. This architecture, called *Two Level Transducer (TLT)* architecture, implements a high-performance FST in two stages, namely *state memory* and *layout*. The states of the high-performance FST are implemented in state memory and layout processing is used to change its state during the processing of a packet (i.e., perform the matching operations required for the inspection processing).

Using a specialized associative memory, called TCAM, for performing the layout processing of the TLT, results in a high processing speed without penalty on memory

usage for the CIE. The amount of TCAM memory needed to store the layout entries, greatly depends on the number of bits processed in each of the states of the high-performance FST. We show that the problem of minimizing the amount of TCAM required for storing the layout entries in TCAM is a difficult problem (i.e., NP-hard [41]). We then present dynamic programming techniques which can be used as a heuristic to reduce the amount of TCAM required for layout processing.

We also introduce a performance criterion, called *worst-case throughput*, and show that this criterion is an appropriate measure for evaluating the speed performance of CIEs. Further, we present a graph-based model for the operation of CIEs and express the worst-case throughput in terms of the parameters of this graph. Using this graph-based model, we transform the problem of calculating the worst-case throughput into a Minimum Weight to Time Ratio (MWTR) problem for which many efficient algorithms exist [42–45].

The outline of this chapter is as follows. Section 3.1 explains the TLT architecture. In Section 3.2 we show that the problem of minimizing the TCAM required for representing a layout entry is NP-hard [41]. Section 3.3 presents the dynamic programming techniques that we have developed for TCAM memory optimization. In Section 3.4 we explain why we need a new criterion for speed evaluation of CIEs and introduce an appropriate criterion together with its calculation strategy. Section 3.5 presents the main results of the experimental evaluations that we have conducted. The Chapter is concluded in Section 3.6.

3.1 Hardware Architecture of High-Performance FSTs

In this section we first explain how to implement a CIE as an FST. We then explain the idea of using a high-performance FST for improving the processing speed of a CIE. Finally, the TLT architecture for implementing a high-performance FST is presented.

3.1.1 Transforming a Set of Policy Rules to a Transducer

We start this section by considering a greatly simplified set of policy rules defined on a packet. We assume that we process the first 6 bits of incoming packets and there are 13 rules in the set of policy rules as shown in Figure 3.1[1]. There are 3 classes in the policy called Cl_1, Cl_2, and Cl_3. The number of rules in each of Cl_1,Cl_2, and Cl_3 is 1, 9, and 3, respectively. There is also an action associated with each class. For instance, the actions can be: REJECT the packet, send the packet to a specific queue, etc. We have used a simple way to present each rule. Each rule is expressed by a string of length 6 over the alphabet $\{0, 1, *\}$. An incoming packet matches a rule if its bits match to the values expressed by the rule. A '*' in a rule is a "don't care" bit meaning that the corresponding bit can be 0 or 1. For example, both packets 000010 and 000011 match to the third rule which is 00001*.

Figure 3.2 shows a directed graph representing these rules. There are 3 different paths on this graph that start from the initial node, labeled by 1, and end at the node labeled by 16. Each of these paths represents one rule of the class Cl_3. Similarly, corresponding to the rules of the other two classes, Cl_1 and Cl_2, there are 1 and 9 paths from the initial node to the nodes labeled by 14 and 15, respectively. This

[1]While a typical set of content inspection policy rules comprises (at least) tens of rules with many bits involved in each rule, we consider this example to facilitate explanation of the transformation process.

Rule No.	Rule	Class
1	000000	Cl_1
2	000001	Cl_2
3	00001*	Cl_2
4	0001**	Cl_2
5	001*10	Cl_2
6	001*0*	Cl_2
7	01**10	Cl_2
8	01**0*	Cl_2
9	1***10	Cl_2
10	1***0*	Cl_2
11	001*11	Cl_3
12	01**11	Cl_3
13	1***11	Cl_3

Figure 3.1: The set of policy rules for an over-simplified content inspection problem.

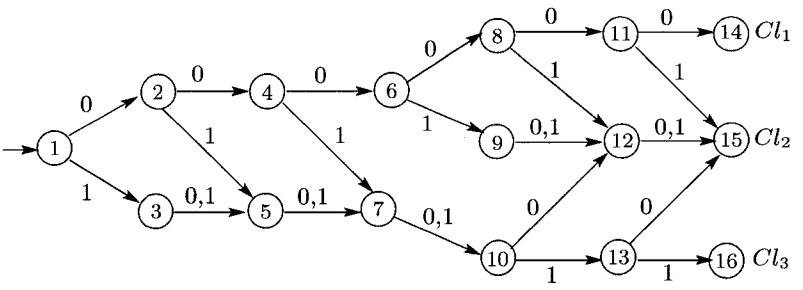


Figure 3.2: The directed graph representation for the policy rules of Figure 3.1 is very similar to state diagram of an FST.

graph is very similar to the state diagram of an FST, with input alphabet $\{0, 1\}$ and output alphabet $\{Cl_1, Cl_2, Cl_3\}$, which can be used for inspecting the packets based on these rules as follows. It starts inspecting a packet with arrival of the first bit, changing its state from the initial state 1 according to this bit. Similarly, in each next state, the transducer uses the corresponding bit of the packet to change its state and this process is continued until the transducer is in one of the states 11, 12, or 13. From each of these states, the transducer reads the sixth bit of the packet and based on its value, it both produces one symbol of its output alphabet (i.e., Cl_1, Cl_2, or Cl_3) which shows the result of inspection, and makes the transition to one of the final nodes 14, 15, or 16.

3.1.2 High-Performance FSTs for Content Inspection

The operations for the state transition of the above-mentioned FST should be completed in the arrival time of a single bit. However, with the usual incoming line speeds to the routers of today (typically several giga bits per second), the arrival time of a single bit is much less than the time of a single memory access operation (a few nano seconds) which is the minimum operation required for a state transition. Our solution to this problem is to use a high-performance FST, in which, each state transition occurs after receiving a group of bits whose arrival time is big enough for the transducer to perform the operations necessary for that state transition. Depending on the speed of communications, in practice the high-performance FST may process 8, 12, 16, or more bits at each state.

Figures 3.3-(a) and 3.3-(b) illustrate the state diagrams corresponding to two high-performance FSTs for the policy of Figure 3.1, which process 2 and 3 bits in each state transition, respectively. An outgoing edge with an input-string label $i - j$ from a state of these high-performance FSTs means that if the decimal value equivalent to the processed bits in that state falls within the range $[i\ j]$, the state would be changed

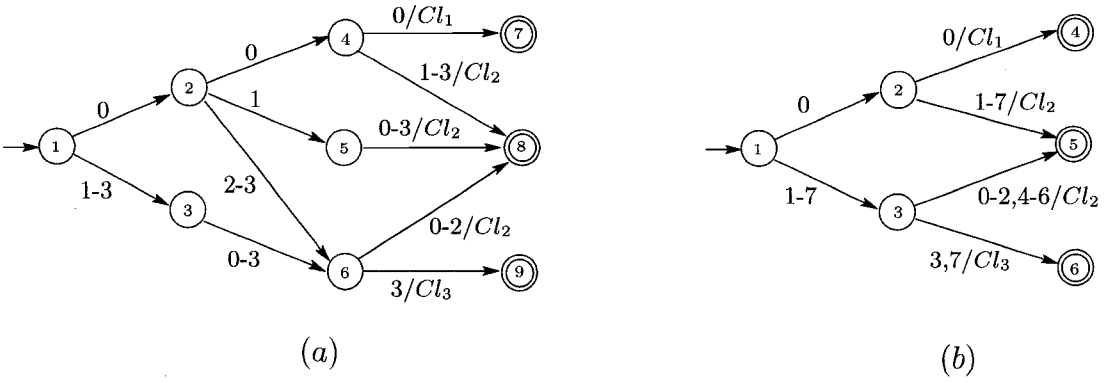


Figure 3.3: The state diagram of a high-performance FST for the policy rules of Figure 3.1 processing, (a) 2 bits, and, (b) 3 bits, in each state.

according to this edge. For example, in state 1 of Figure 3.3-(b), if the first 3 bits of the packet are 000, the state changes from 1 to 2. Otherwise, the state changes from 1 to 3. Note that on transitions to their final nodes, the transducers generate one of the output symbols Cl_1, Cl_2 , or Cl_3 as the result of inspecting each packet.

A state of the high-performance FST can be easily implemented in a lookup table having a number of addresses, or in any other form of address conversion. In the lookup table implementation, upon receiving the specified number of data bits, only a single memory access is required for determining the next memory address of the lookup table. As an example, consider a high-performance FST that processes 8 bits in each state and suppose that a current table address is loaded in the high order bits of a register. Then 8 bits from the data stream are loaded into the low order bits of the register and act as an offset 0-255. Once loaded, data at the location indicated by the register is loaded into the higher order bits. It is first checked whether it is an action determining the result of content inspection (the time required for this checking process is negligible in comparison with the memory access time). If it is not an action, the next 8 bits from the data stream are loaded into the low order bits to form a new address. This process continues until a final state is reached (i.e.,

an action is loaded into the higher order bits of the register), which completes the inspection of a packet. A state transition of the high-performance FST with this implementation, is performed in a single memory access time.

At the line speeds of several gigabits per second, a large number of bits should be processed in each state of the high-performance FST. This makes the lookup table storage of the states very large and results in increased cost and reduced performance by forcing the use of external memory devices instead of fast embedded memory. Therefore, there is a practical limitation on the number of bits that may be processed in parallel in such a high-performance FST. In order to overcome this limitation, we introduce the TLT architecture for implementing the high-performance FSTs which turns out to be very efficient for memory optimization.

3.1.3 Two-Level Transducer Architecture

A TLT implements a high-performance FST in two stages, *state memory* and *layout*, hence the name. State memory is used to implement the states of the high-performance FST. Each state is implemented in a *state memory node*. State memory nodes contain data that the high-performance FST uses to select and execute a sequence of inspection steps. This data can be an indication that the inspection is done and an action is produced. Otherwise, the data contains the number of bits to be read from the input stream, and the address of a *layout entry* that processes these input bits. Each layout entry contains data that encodes the incoming-string labels on the outgoing edges of a state and also an *offset* for each label.

A state memory node reads a specified number of bits from the input and sends them to the layout entry (specified by the address contained in it) for performing layout processing. The layout entry determines the edge-label that matches the input bits in that state and returns the offset of this label to the calling state memory node. This offset is used for reading a single word from within the calling state memory node

index	
0	00001
1	00010
2	01011
3	010*0
4	01***
5	*****

Figure 3.4: A TCAM of width 5 containing 6 rules.

(i.e., the current state of the high-performance FST), which determines the next state memory node (i.e., the next state of the high-performance FST). This procedure of reading a number of bits from the input packet, sending them to a layout entry for layout processing, and choosing the next state memory node is repeated until a state memory node that contains an action is reached. At this time the inspection of the current packet is completed and the inspection of the next incoming packet begins.

State memory nodes can be implemented in either dynamic or static RAM. A well-suited medium for performing the layout processing is Ternary Content Addressable Memory (TCAM) [14,46]. A TCAM stores a number of strings of a fixed length over the three-valued alphabet $\{0, 1, *\}$. Each of the stored strings is called a TCAM *rule* and there is an index associated to each rule showing the position of the rule inside the TCAM. For example, Figure 3.4 shows a TCAM with six rules of length 5. A TCAM compares a given input binary string, called an input *key*, against all of its rules in parallel and returns the smallest index among the indices of the rules that match the input key. In this comparison, a "don't care" symbol, $*$, in a rule means that the corresponding binary value in the input key can be 0 or 1. The set of rules stored in a the TCAM of Figure 3.4 covers all the possible binary strings of length 5; This is guaranteed by putting the "catch all" rule as the last rule of the TCAM.

The layout entry corresponding to a state of a high-performance FST can be

realized in a TCAM unit by encoding the labels of the outgoing edges of that state as rules of the TCAM. The input bits read by the corresponding state memory node will be used as the input key for the TCAM unit and the state memory node uses the index returned by the TCAM as an offset. Figure 3.5 shows the high-performance FST of Figure 3.3-(b) together with its TLT implementation using TCAM for realizing the layout entries. The resulted TLT contains six state memory nodes (SM_1 to SM_6), corresponding to the six states of the high-performance FST. State memory entries SM_1, SM_2, and SM_3 refer to TCAM units, and SM_4, SM_5, and SM_6 are final states, in which the inspection of a packet is completed.

This example shows how we can exploit the dependencies of rules by the TLT architecture; since the outgoing edges from states 1 and 2 of the high-performance FST have the same labels, we can implement the set of these labels only once in the layout (here TCAM) and refer to it from the state memory nodes corresponding to states 1 and 2. The TCAM entry T_1 implements this set and is referred to from both SM_1 and SM_2. T_1 contains two rows to encode this label set with the first row having priority over the second row. This means that the TCAM first matches the 3-bits input sequence against "000". If there is a match, it returns the index corresponding to this label. Otherwise it returns the index corresponding to the second row. This index is used as an offset by the calling state memory nodes (SM_1 or SM_2) to determine the next state memory node.

The time of a state transition in the TLT implementation of a high-performance FST is dominated by its layout processing time plus the time of the single state memory access performed after getting the offset from the layout entry. Using TCAM for performing layout processing provides a considerable boost in the inspection speed by matching the read input string against all of its rows (i.e., edge labels of the high-performance FST) in parallel. If we perform layout processing in TCAM, the

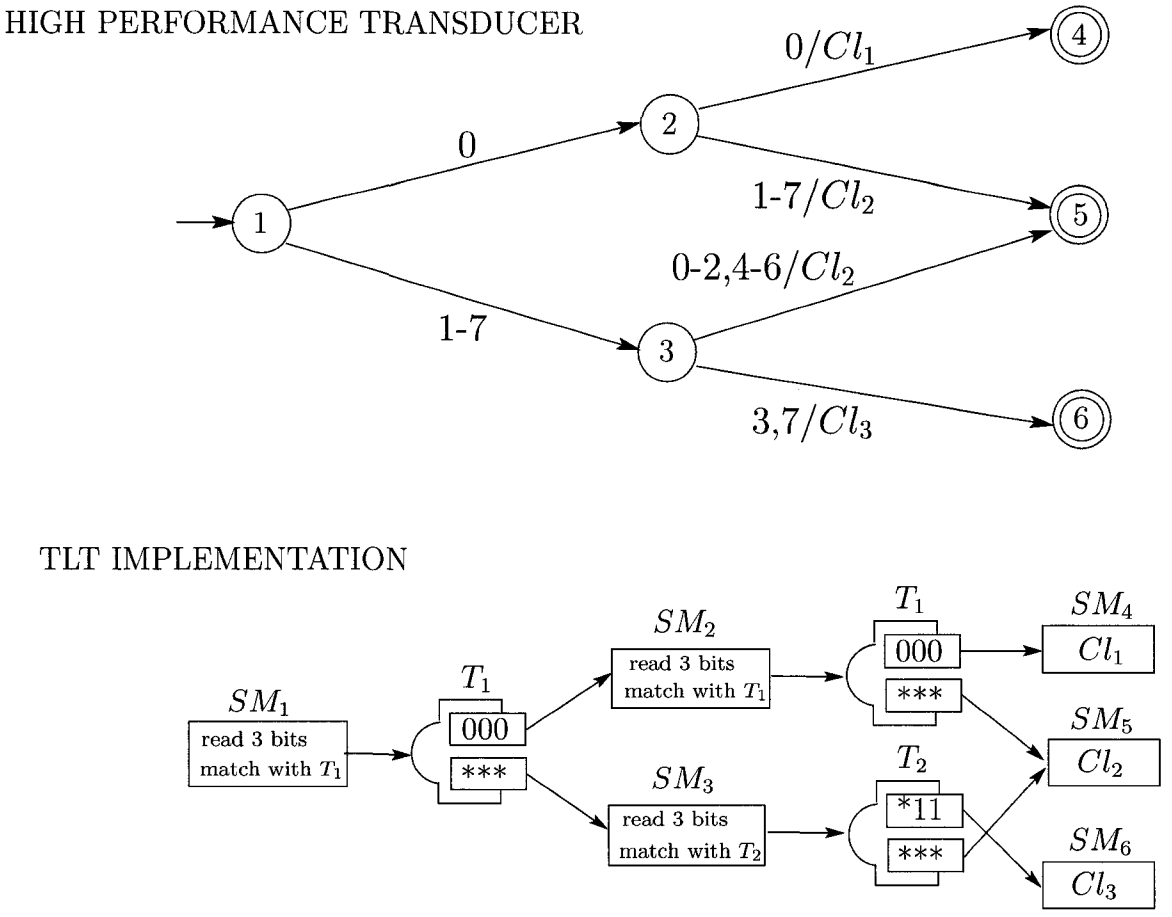


Figure 3.5: The state diagram of a high-performance FST and its TLT implementation for the policy rules of Figure 3.1.

processing time of a layout entry would be almost the same as a single state memory access time. Also, since state memory is separate from the layout processing medium, we can perform the inspection of two packets in parallel by simultaneously performing the layout processing for one packet and accessing state memory for the other. This means that the state transition time in the TLT implementation can be reduced to one memory access time which is the same as that of the lookup table implementation. Nonetheless, the memory requirement of the TLT implementation may be exponentially smaller than that of the direct lookup table implementation. Note that in the lookup table implementation, the string of read input bits at each state is used to index the lookup table from its current location. In the TLT implementation, however, this string is first translated to an offset (by layout processing), which is significantly smaller than the string, and then used for indexing the state memory. In fact, it is this exponential reduction in the size of the required memory which enables the TLT architecture to implement high-performance FSTs with large number of processed bits at each state.

A typical CIE built as a high-performance FST with the TLT architecture needs only two modules for the state memory and layout, and a register that points to a state memory node corresponding to the current state of the high-performance FST during the inspection process. The initial state memory node contains a given number of bits to be read from the input packet and a reference to a layout entry that inspects these bits. Starting from here, the given number of bits are read from the input packet and sent to the TCAM for lookup. The TCAM returns an offset to the calling state memory node which uses it to determine the address of the next state memory node to be loaded into the register. This cycle, i.e., reading bits, matching them against a TCAM unit, and loading the next state memory node, continues until a state memory node containing a final action is reached. Figure 3.6 shows the flow of information in such an architecture. The state transition time of this CIE is two memory access

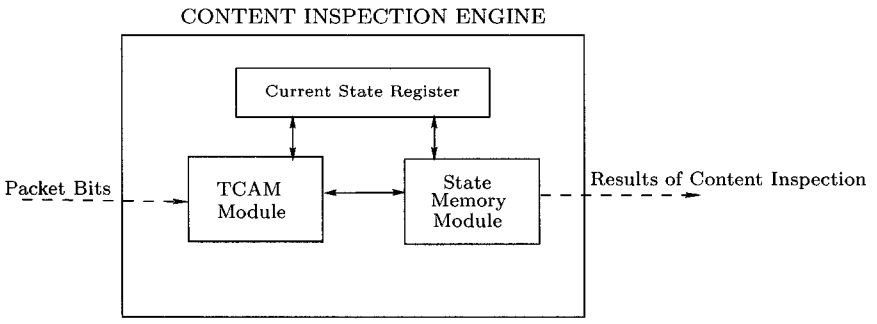


Figure 3.6: Structure of a CIE based on the TLT architecture.

time. Adding a second current state register to this architecture for processing two packets in parallel, reduces this time to one memory access time.

3.2 What is the Complexity of the Minimal TCAM Representation Problem?

TCAM is an appropriate medium for implementing layout entries of the TLT architecture as it provides a considerable boost in the processing speed of the CIE by allowing all of its entries to be read in parallel. However, increasing the number of processed bits in a single state of a high-performance FST beyond 10 bits, may tend to a large increase in the number of the corresponding TCAM entries. The problem of finding the minimal TCAM representation of a subgraph is a challenging problem. In this section we show that the Minimal Normal Disjunctive Form problem [41] can be polynomially reduced to this problem. This transformation implies that the minimal TCAM representation problem is NP-hard. We also present a heuristic that we have used for calculating the number of TCAM rules corresponding to a layout entry.

3.2.1 Minimal Disjunctive Normal Form Problem

Disjunctive Normal Form is a method of standardizing logical statements. A logical statement is in Disjunctive Normal Form (DNF) if it is a disjunction (sequence of ORs) of clauses, where a clause is a conjunction (AND) of one or more literals (i.e., statement letters, and negations of statement letters). The only propositional operators in DNF are AND, OR, and NOT which are denoted by $\wedge$, $\vee$, and $\neg$, respectively. The NOT operator can only be used as part of a literal, i.e., it can only precede a propositional variable. For example, the following logical statements are in DNF:

$$A$$

$$\neg A \vee B$$

$$(A \wedge B) \vee \neg C$$

$$(A \wedge B \wedge \neg C) \vee (\neg B \wedge C) \vee (A \wedge \neg C)$$

Every logical statement consisting of a combination of multiple $\wedge$, $\vee$ and $\neg$s can be converted to DNF. Such a conversion involves using logical equivalences, such as the double negative law, De Morgan's laws, and the distributive law. Note that in some cases, conversion of a statement to DNF can lead to an exponential explosion of the statement. For example, the DNF of the following statement has 2^n terms:

$$(X_1 \vee Y_1) \wedge (X_2 \vee Y_2) \wedge \ldots \wedge (X_n \vee Y_n)$$

The problem of finding a minimal DNF of a logical statement is an NP-complete problem [41]. This problem is stated as follows:

Minimal Disjunctive Normal Form Problem

Instance: Set $U = \{u_1, u_2, ...u_n\}$ of variables. Set $A \subseteq \{T, F\}^n$ of "truth assignments" and a positive integer K.

Question: Is there a DNF expression E over U, having no more than K disjuncts, such that E is true for precisely those assignments in A and no others?

3.2.2 The Relation between the TCAM Representation of Layout Entries and their DNF

The TCAM representation of the subgraph emanating from a state of high-performance FST can be obtained by applying a chosen heuristic (we will present one such heuristics later in this section) to the subgraph. For example, consider the subgraph of Figure 3.7. The TCAM representation of this subgraph, obtained by simply reading all the paths of the subgraph from top to the bottom, is:

0011
01*1
1111
110*
1011
1000

It is not difficult to see that we can express a TCAM by a DNF logical statement. Consider the ith bit in a row of a TCAM. We represent this bit by X_i and $\neg X_i$ if this bit is equal to 1 and 0, respectively. Using this notation, we can express each row of the TCAM by a conjunctive term. For example, the first and the second rows of the above TCAM can be expressed by $\neg X_0 \wedge \neg X_1 \wedge X_2 \wedge X_3$ and $\neg X_0 \wedge X_1 \wedge (X_2 \vee \neg X_2) \wedge X_3$, respectively. Note that for the second row, the value of the third bit does not affect the outcome of matching the input sequence against this row and thus we can simplify its corresponding expression to $\neg X_0 \wedge X_1 \wedge X_3$. The DNF logical statement corresponding to the TCAM, obtained by disjointing the conjunction terms of its rows, is:

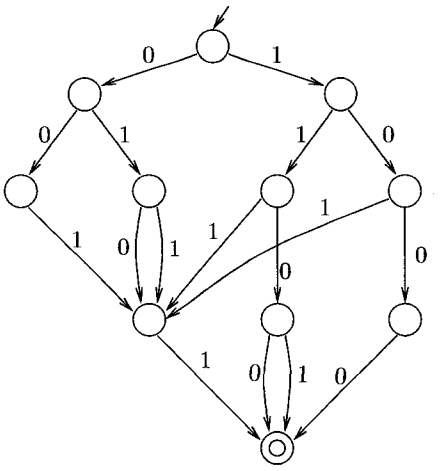


Figure 3.7: A sample subgraph

$$[\neg X_0 \wedge \neg X_1 \wedge X_2 \wedge X_3] \vee [\neg X_0 \wedge X_1 \wedge X_3] \vee [X_0 \wedge X_1 \wedge X_2 \wedge X_3] \vee$$

$$[X_0 \wedge X_1 \wedge \neg X_2] \vee [X_0 \wedge \neg X_1 \wedge X_2 \wedge X_3] \vee [X_0 \wedge \neg X_1 \wedge \neg X_2 \wedge \neg X_3]$$

3.2.3 Complexity of the Minimal TCAM Representation Problem

The correspondence between a TCAM and its DNF logical statement can be used to show that the problem of finding the minimal TCAM representation of a subgraph is an NP-hard problem. For this purpose, we show that the minimal DNF problem can be polynomially reduced to the minimal TCAM representation problem.

Let us assume that $U = \{u_1, u_2, ...u_n\}$ is a set of variables and $A \subseteq \{T, F\}^n$ is a set of truth statements. The minimal DNF problem determines whether there is a DNF expression E over U, having no more than K disjuncts, such that E is true for precisely those assignments in A and no others. We use a polynomial time algorithm to build a transducer representation of set A, i.e., every path from the initial node to the final node of this FST corresponds to one truth assignment of set

A. If we can find the minimal TCAM representation of this FST, then using the correspondence between a TCAM and its DNF logical statement, we can transform this minimal TCAM representation to a statement which is the minimal DNF for set A. Therefore, we have been able to polynomially reduce the NP-complete problem of minimal DNF to the minimal TCAM representation problem which implies that the minimal TCAM representation problem is an NP-hard problem. Please note that the size of the FST built over set A is linear with respect to $|A|$. However, the size of the language obtained from an FST could be exponential with respect to the size of the FST.

Due to the NP-hard nature of the problem, we should try to find efficient algorithms that yield a small TCAM representation of the subgraph with an acceptable running time complexity. We introduce one heuristic for this purpose in the following subsection.

3.2.4 A Heuristic for Realization of Layout Entries in TCAM

Consider a state S_1 of a high-performance FST with k next states, i.e., S_1 has k outgoing edges, each edge labeled with one or more input strings, that end at k different next states. A fast heuristics for encoding the labels of the outgoing edges of S_1 in TCAM works as follows. It first builds k minimal state machines, each state machine representing the input strings on one outgoing edges of S_1. Then it generates the TCAM rules corresponding to each outgoing edge by enumerating all paths on its corresponding state machine, considering all double-labeled edges as stars ($*$).

Example 3.2.1. *Let us use this heuristic to generate the TCAM rules corresponding to a state S_1 of an imaginary high-performance FST as illustrated in Figure 3.8-(a). Four input bits are processed at S_1 and S_2, S_3, and S_4 are the 3 next state of S_1, i.e., $k = 3$. Figure 3.8-(b) shows a minimal state machine representation of all the*

labels of the outgoing edges of S_1. *Figure 3.9 presents* 3 *minimal state machines each representing the labels on one outgoing edge of* S_1. *The set of TCAM rules generated by enumerating all the paths that end at* S_2, S_3, *and* S_4 *are equal to* $\boxed{\begin{array}{l}000*\\0010\\01*0\\1*10\end{array}}$, $\boxed{\begin{array}{l}0011\\01*1\\10*1\end{array}}$, *and* $\boxed{\begin{array}{l}1000\\110*\end{array}}$, *respectively.* ◀

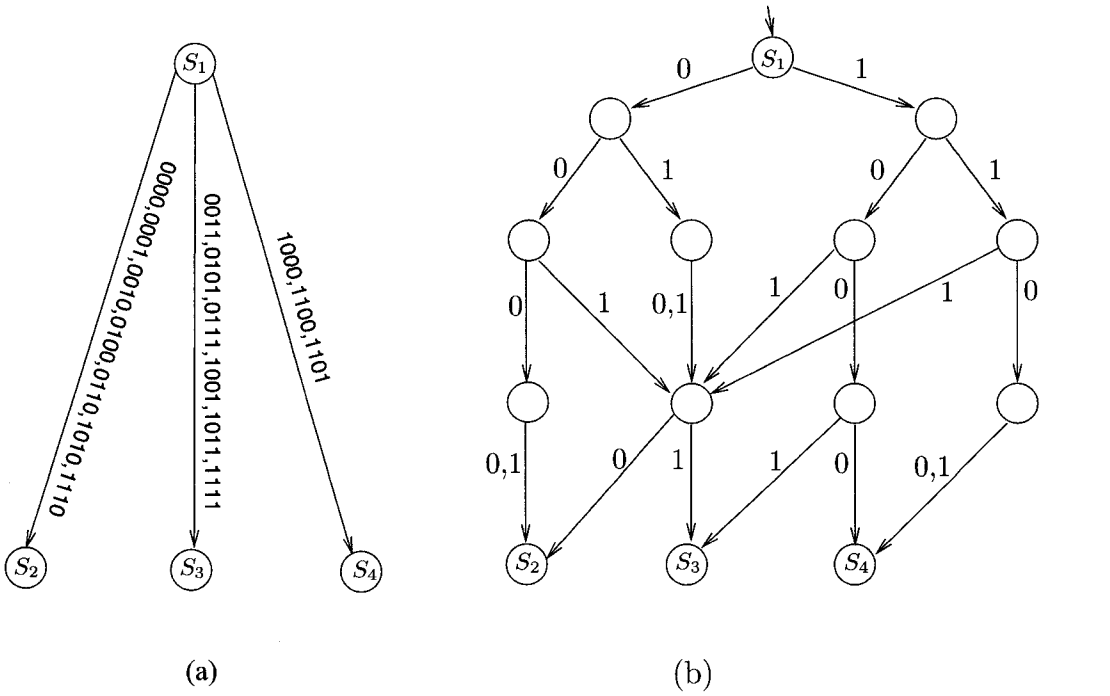


Figure 3.8: (a)- A state of a high-performance FST and its next states. (b)- The minimal state machine representation of the labels of the outgoing edges of S_1.

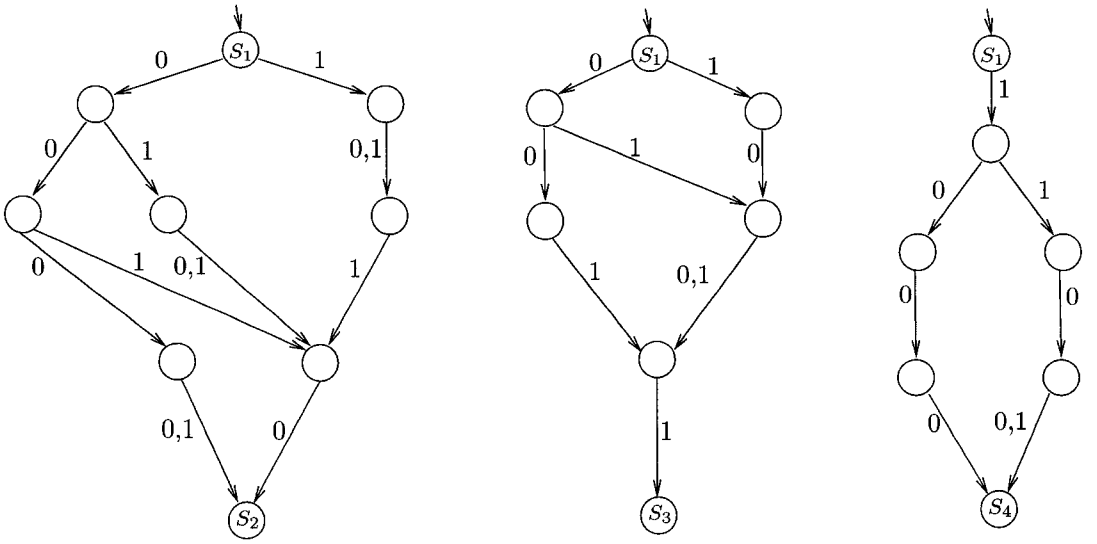


Figure 3.9: The minimal state machines corresponding to the labels of outgoing edges of S_1, from Figure 3.8, ending in the next states S_2, S_3, and S_4.

3.3 TCAM Optimization Using Dynamic Programming Techniques

As it was mentioned in Section 3.1, in the TLT architecture repetitive layout entries can be realized once and referred to, as needed, from the state memory nodes. Realizing the layout entries in TCAM has the extra advantage of performing the matching operation for each state in TCAM units whose very fast access speed and parallel structure makes wire-speed processing possible. However, TCAMs are still expensive types of memory, even with all recent developments that have made them more affordable [47]. The amount of TCAM memory needed to store the layout entries, greatly depends on the number of bits processed in each of the states of the high-performance FST. Determining the number of bits to be processed at each state of the high-performance FST can be done in many different strategies, where choosing one strategy over another may yield very different results in terms of TCAM size and

content inspection speed.

Example 3.3.1. *Let us consider again the set of policy rules of Figure 3.1. Suppose that we are looking for a high-performance FST for these rules that needs the minimum number of TCAM bits for its layout entries and satisfies the following requirement: it should process, in average, 2 or more bits in each state transition of every path that starts from the initial node and ends at a final node. There are several high-performance FSTs which yield an average number of 2 or more processed bits in each state transition. For instance, both of the high-performance FSTs of Figure 3.3 meet this requirement.*

The high-performance FSTs of Figure 3.3-(a) and Figure 3.3-(b) are implemented in Figure 3.10 and Figure 3.5, respectively. The TLT of Figure 3.10 needs 16 TCAM bits for implementing its unique layout entries (T_1, T_2, T_3, and T_4) which is more than the 12 TCAM bits required for the TLT of Figure 3.5. This is while the high-performance FST of Figure 3.3-(a) has a smaller average number of processed bits. In general case, however, we may also get the opposite situation, depending on the topology of the set of policy rules. For instance, another high-performance FST that processes, in average, 3 bits in each state transition is shown in Figure 3.11 together with its TLT implementation. It needs 22 TCAM bits for implementing its unique layout entries (T_1, T_2, T_3, and T_4) which is more than the amount of the TCAM required for the TLT of Figure 3.10. ◀

For practical situations, where each rule of the policy involves many bits and there are many rules in the policy, an automated procedure is required to minimize the amount of TCAM by exploiting rule dependencies efficiently. In this section we explain one such memory optimization procedure based on dynamic programming techniques. This procedure determines the number of bits that should be processed in each state of the high-performance FST so that the inspection speed of the CIE is

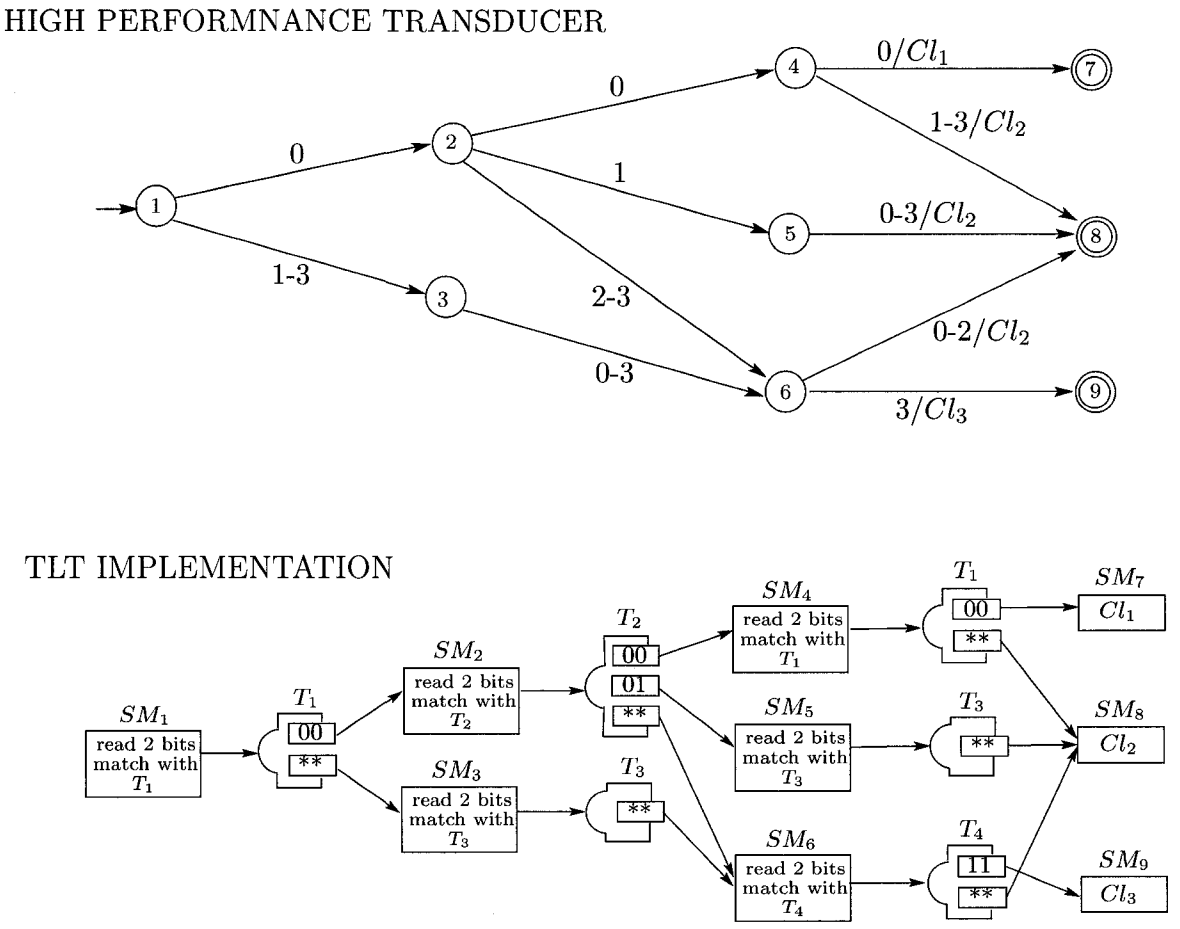
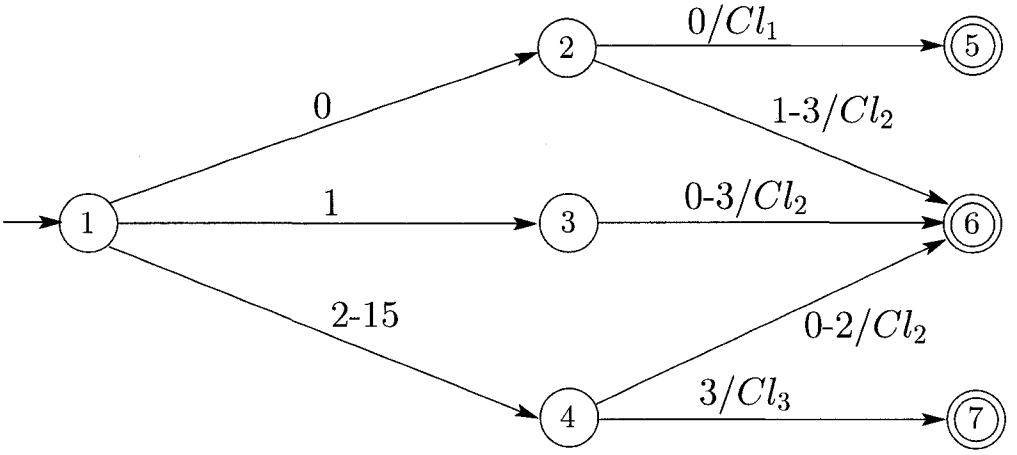


Figure 3.10: State diagram of a high-performance FST and its TLT implementation for the policy rules of Figure 3.1.

HIGH PERFORMANCE TRANSDUCER



TLT IMPLEMENTATION

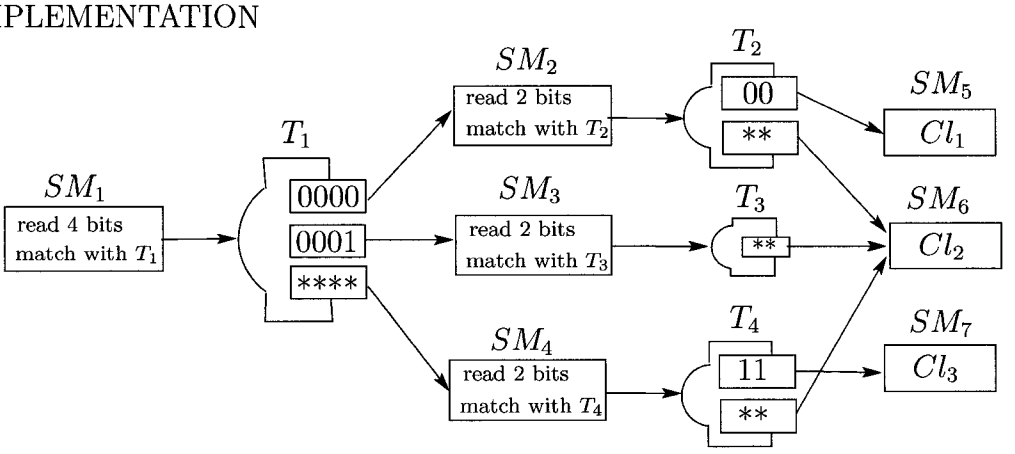


Figure 3.11: State diagram of a high-performance FST and its TLT implementation for the policy rules of Figure 3.1.

not smaller than a given speed and the dependencies between the rules are efficiently extracted.

The initial research work towards development of dynamic programming techniques that are presented in the following has been conducted by Andrej Pelc and Wojciech Fraczak, from the research group of IDT Canada, for optimizing the memory requirements of the *PAX.port* engines of IDT Canada [48] which have a one-level memory architecture. We have adopted these techniques and have generalized them for optimizing the memory requirements of the TLT architecture as presented here.

3.3.1 Problem Statement

We start the explanation of the dynamic programming techniques assuming that we have a directed acyclic graph with one-bit labels (i.e., a transducer with input alphabet $\{0,1\}$) representation for the input set of the policy rules of a specific content inspection problem. We call this input graph an *elementary graph.* In this graph, nodes of out-degree 0 are called *leaves*: each of them is labeled by an action corresponding to one rule of the policy rule set. Other nodes are called *internal nodes.* For any internal node u, all nodes u' such that there exists a directed path of length k from u to u' are called *k-descendants* of u. Any k-descendant u' of u corresponds to a binary sequence of length k. The same node u' can correspond to many binary sequences, and thus it may be a k-descendant and an l-descendant of u, for different k and l. In particular, nodes v and w such that (u,v) and (u,w) are directed edges in the graph, are called *children* of u. On an elementary graph, one child of u corresponds to the input bit 0 and the other to the input bit 1 of the processed binary string. For any node v, the subgraph of G induced by v and all descendants of v is denoted by G_v.

Consider a fixed internal node u of the elementary graph. We say that changing symbol size at u from 1 to k is *legal*, if the following conditions are satisfied:

- $k \leq B$, where B is a positive integer constant given *a priori*, called the *symbol size bound.*

- for every $k' < k$ there does not exist a k'-descendant of u which is a leaf whose label is a non-reject action of the classifier.

A legal change of symbol size at u from 1 to k results in the following transformation of the graph. Consider all binary sequences of length k. For any such sequence s add a *new* directed edge starting at u. The child node at the end of such an edge, is either the node corresponding to s or a (new) leaf with label *reject* in the case when there is no node corresponding to s in the original graph.

Intuitively, changing symbol size to k at node u causes reading an entire string of k bits in one step, thus potentially speeding up the processing. In practice, we have a finite set of possible choices (for example $\{1, 4, 8, 16, 32, 64\}$) for the lengths of the TCAM rules used in the TLT implementation of the high-performance FST. We will refer to this set as "the set of available symbol sizes". The first condition of legality is imposed by the maximum available symbol size.

Consider the graph G' resulting from an elementary graph G by applying a sequence of legal symbol size changes at some internal nodes of G. More precisely, nodes of graph G' consist of some nodes of G and of new nodes with label *reject.* Edges of G' are old edges that have not become new ones and *new* edges added during all symbol size changes. G' represents the same set of policy rules as G and we say that G' is *equivalent* to the elementary graph G.

The graph G' can be considered as the state diagram for a high-performance FST. For a node u in G', we denote by $N_{TCAM}(u)$ the number of rules used for TCAM representation of its corresponding layout entry in the TLT implementation. This number can be obtained from using a heuristic such as the fast heuristic of the previous section. The *cost* of a node u in G', denoted by $c(u)$, is defined as the

summation of cost of its children and $N_{TCAM}(u)$. The *cost* of the graph G' is defined as the cost of its root node, i.e., if r is the root of G' then $c(G') = c(r)$. In other words, the cost of G' is equal to the total number of TCAM rules required for implementing the layout entries of its corresponding high-performance FST in TCAM. Having these definitions, the problem can be formulated as follows:

Symbol-Size Selection Problem- Given an elementary graph G, a set of available symbol sizes, and a positive integer K, find a minimum-cost graph G', equivalent to G, such that the average symbol size on every path of G' is greater than or equal to K. □

The value of K, given as an input of the problem, is determined by the incoming line speed to the CIE, access time of the state memory, and TCAM access time. For example, if we store state memory nodes in a DRAM with access time of 4 (nsec) and the TCAM has an access time of 6 (nsec), then each state transition of the high-performance FST is performed in 10 (nsec). This means that for a CIE with target speed of 1 Gbits/sec, the value of K, i.e., the "average" number of bits processed at each state of the high-performance FST, is 10.

3.3.2 Optimal Solution for Trees

We proceed with presenting an algorithm which finds a provably optimal solution of our main problem in the special case when the given elementary graph G is a tree. The algorithm is based on the method of dynamic programming. The main idea underlying this algorithm will be used in the heuristic that we will use for arbitrary acyclic graphs.

We start by presenting an algorithm which solves the following simpler problem:

Unconstrained Symbol-Size Selection Problem for Trees- Given an elementary tree T and a set of available symbol sizes, find a tree equivalent to T whose cost is minimum. □

Thus we temporarily remove the constraint on the average symbol size of every path of the resulting tree.

Unconstrained Symbol Size Selection Algorithm for Trees

The algorithm works bottom-up. Consider an elementary tree T. For any node u of T, let T_u denotes the subtree of T rooted at u. We will preserve the following invariant for every node v in T.

- a minimum-cost tree T'_v equivalent to T_v is computed and its cost c_v is known.

For any leaf v, the invariant is easy to satisfy: the minimum-cost tree consists of the unique node v, and its cost is 0. Let v be any internal node of T and suppose that the invariant is preserved for all descendants of v. For any legal symbol size k at v, consider the set $S_v(k)$ of all nodes of T which are k-descendants of v. Let $T_v(k)$ be the tree rooted at v and defined as follows:

> there are $|S_v(k)|$ directed edges from v going to all nodes of $S_v(k)$ and $2^k - |S_v(k)|$ directed edges from v going to new leaves with label *reject*. To each node $u \in S_v(k)$ attach the tree T'_u rooted at u.

The cost of $T_v(k)$ is $c(T_v(k)) = N_{TCAM}(v) + \sum_{u \in S_v(k)} c_u$, and is upper bounded by $|S_v(k)| + \sum_{u \in S_v(k)} c_u$. Let s_v be equal to the integer k for which the above cost is minimum, where the minimum is taken over all legal symbol sizes at v. Define T'_v to be $T_v(s_v)$ and c_v to be $c(T_v(s_v))$. .

Let r be the root of T. The tree T'_r is the output of the algorithm. Now traversing T'_r top-down from r, symbol sizes at each node can be retrieved. □

Theorem 3.3.1. *The unconstrained symbol size selection algorithm for trees finds a minimum-cost tree equivalent to any given elementary tree. It works in time $O(n)$, where n is the size of the input tree.*

Proof. If T'_u is a minimum-cost tree equivalent to T_u for any descendant u of v, then the tree T'_v computed by the algorithm is a minimum-cost tree equivalent to T_v. The algorithm uses constant time at any node of the tree (recall that the upper bound B is constant), hence its total execution time is linear. □

We now return to our main problem which is cost-optimization under constraints on the average symbol size of every path of the tree:

Symbol-Size Selection Problem for Trees- Given an elementary tree T, a set of available symbol sizes, and a positive integer K, find a minimum-cost tree T', equivalent to T, such that the average symbol size on every path of T' is greater than or equal to K. □

We will solve this problem using a refinement of the above algorithm for the unconstrained problem. The high-performance FST generated using the symbol sizes provided by this algorithm, must satisfy the required average symbol size restriction. Thus every time a symbol size is chosen, we must calculate the effect of that choice on our effective symbol size for the entire conversion. This is done by keeping track of the generated *surplus* for each symbol size choice. For each legal symbol size choice of k, the generated surplus d is calculated as $d = k - K$.

The whole idea of this dynamic programming algorithm is to allow a negative surplus (i.e., deficit) at certain areas of the graph, where smaller symbol sizes are more beneficial, to be recovered by the use of bigger symbol sizes at other areas of the graph. This requires every node v in the graph to have an accompanying list of alternatives available to guarantee a specific surplus d requirement. Intuitively alternatives guaranteeing large positive surplus will use bigger symbol sizes and thus use more memory, while alternatives guaranteeing negative surplus will use smaller symbol sizes and less memory.

Symbol Size Selection Algorithm for Trees

The algorithm works bottom-up. Consider an elementary tree T. For any node v of T, T_v denotes the subtree of T rooted at v. Also for such a node v, L_v denotes the list of triples (d, c_v^d, k), where d is a guaranteed surplus in v, c_v^d is the cost associated with this surplus (will be defined later), and k is a legal symbol size that guarantees this surplus in v. Since a surplus d can be satisfied by any surplus $d' \geq d$, for any two entries $(d_1, c_v^{d_1}, k_1)$ and $(d_2, c_v^{d_2}, k_2)$ of L_v, where $d_1 \leq d_2$ and $c_v^{d_1} \geq c_v^{d_2}$, the first entry can be deleted as the second entry provides a better alternative. We will implement this observation while constructing the list in the following.

For any leaf v, L_v is $\{(0,0,0)\}$. For every internal node v of T, assuming that the lists corresponding to all descendants of v have been computed, the following information must be computed for generating L_v:

- For every integer k corresponding to a legal symbol size at v, let $D_v(k)$ be the union of surpluses guaranteed by all k-descendants of v shifted by $k - K$:

$$D_v(k) = \{d + k - K \mid \forall u \in S_v(k), \exists (d', c_u^{d'}, k') \in L_u \text{ such that } d' \geq d \text{ , and } d' = d \text{ for at least one } u \in S_v(k)\}.$$

- The set of all possible surpluses at v, denoted by D_v is obtained as follows:

$$D_v = \bigcup \{D_v(k) \mid k \text{ is a legal symbol size at } v\}$$

- For every surplus $d \in D_v$, a tree T_v^d guaranteeing a surplus of d at v and equivalent to T_v is computed; a positive integer c_v^d, called the *cost* of T_v^d, and a corresponding legal symbol size are known (as explained in the following). The point (d, c_v^d, k) must now be added to L_v.

- for any two entries $(d_1, c_v^{d_1}, k_1)$ and $(d_2, c_v^{d_2}, k_2)$ of L_v where $d_1 \leq d_2$ and $c_v^{d_1} \geq c_v^{d_2}$ the first entry is deleted from the list.

Note: For some d and v there may be no tree equivalent to T_v that guarantees a surplus of at least d. In this case T_v^d is undefined.

Computing T_v^d and c_v^d .

For any legal symbol size k at v, denote by $S_v(k)$ the set of all nodes of T which are k-descendants of v. Fix symbol size k at node v. For every $d \in D_v(k)$, let $T_v^d(k)$ be the tree rooted at v and defined as follows:

> there are $|S_v(k)|$ directed edges from v going to all nodes of $S_v(k)$ and $2^k - |S_v(k)|$ directed edges from v going to new leaves with label *reject*. To each node $u \in S_v(k)$ attach the tree $T_u^{d+(K-k)}$ rooted at u, if existed, and $T_u^{d'+(K-k)}$ with smallest cost among all the trees with $d' \geq d$, if otherwise.

The cost of $T_v^d(k)$ (if this tree is defined) is:

$$c(T_v^d(k)) = N_{TCAM}(v) + \sum_{u \in S_v(k)} c_u.$$

where:

$$c_u = \begin{cases} c(T_u^{d+(K-k)}) & \text{if } T_u^{d+(K-k)} \text{ exists} \\ c(T_u^{d'+(K-k)}) & \text{Otherwise; } T_u^{d'+(K-k)} \text{ has the smallest cost among all the trees with } d' \geq d \end{cases}$$

For a fixed d, let s_v^d be equal to the integer k for which the above cost is minimum, where the minimum is taken over all k corresponding to legal symbol sizes at v. Define T_v^d to be $T_v^d(s_v^d)$ and c_v^d to be the cost of this tree. If $T_v^d(k)$ is not defined for any k then T_v^d is not defined.

Let r be the root of T. The tree $T_r^d, d \geq 0$, with minimum c_r^d is the output of the algorithm[2] . Now traversing the output tree top-down from r, symbol sizes at each

[2]Note that any tree T_r^d with $d > 0$, guarantees an average symbol size greater than K on all its paths.

node can be retrieved. If no tree $T_r^d, d \geq 0$, is defined, the problem does not have any solution for the given input T and K. □

Theorem 3.3.2. *Let T be an elementary tree for which there exists an equivalent tree such that the average symbol size of every path of it is greater than or equal to K. The symbol size selection algorithm for trees finds a minimum-cost tree equivalent to T with an average symbol size greater than or equal to K on all of its paths. The algorithm works in time $O(nK)$, where n is the size of the input tree.*

Proof. Fix a node v of T and a surplus $d \in D_v$. Suppose that for any descendant u of v, and any entry $(d', c_u^{d'}, k_1)$ in L_u, $T_u^{d'}$ is a minimum-cost tree with an average symbol greater than or equal to $K + d'$ on all of its paths, equivalent to T_u. Then the tree T_v^d computed by the algorithm is a minimum-cost tree, with an average symbol size greater than or equal to $K + d$ on all of its paths, equivalent to T_v. If there exists a tree of average symbol size greater than or equal to K on all of its paths equivalent to T, then the tree chosen by the algorithm has minimum cost among such trees. The algorithm uses time $O(K)$ at any node of the tree, hence its total execution time is $O(nK)$.

□

3.3.3 A Heuristic for Arbitrary Directed Acyclic Graphs

It seems unlikely that a straightforward modification of the algorithm presented in previous section could give an optimal solution in the case of arbitrary directed acyclic graphs. Our intuitive reason for this prediction can be seen already in the simpler case of cost minimization without constraints on the average symbol size; In the case of trees, finding a minimum-cost subtree rooted at a given node v is not difficult, provided that minimum-cost subtrees at descendants of v are already known. As we have seen, it is enough to try all possible symbol sizes at v, for each of them add costs

of respective subtrees plus the number of rules used for TCAM representation of its corresponding layout entry, and choose the symbol size corresponding to the smallest sum. This is due to the fact that, for trees, minimum costs add up. The situation is significantly different in the case of arbitrary directed acyclic graphs.

Consider an elementary graph G with a node v for which we intend to establish optimal symbol size. Suppose that we try symbol size k at v, and that $S_v(k)$ is the set of k-descendants of v. Also suppose that we have already computed minimum-cost graphs G'_u equivalent to G_u, for all $u \in S_v(k)$. Unlike in the case of trees, there is no simple way of using this information to compute the cost of the optimal graph G'_v equivalent to G_v: summing up the costs of $G'_u, u \in S_v(k)$, does not help because the cost of G'_v heavily depends on how strongly graphs $G_u, u \in S_v(k)$, intersect (in the case of trees the respective subtrees are all pairwise disjoint and thus simple addition of costs works). On the other hand, trying to compute the cost of G'_v by exhaustive search is impractical because of exponential complexity of such an approach. In fact it seems plausible that no efficient optimal solution of our problem exists for arbitrary directed acyclic graphs.

Hence for arbitrary directed acyclic graphs we adopt a heuristic approach. The heuristic is based on the idea underlying the symbol size selection algorithm for trees: choosing the symbol size at each node of the graph based on the results of computations concerning descendants of that node. The main missing component is that of estimating how strongly subgraphs at descendants of v intersect. This estimation is done in a preprocessing phase.

Heuristic-Based Symbol Size Selection Algorithm

Consider an elementary graph G, a set of available symbol sizes, and a designated integral average symbol size, K, on every path of G. The algorithm works in two phases: the *preprocessing* phase in which some numerical values characterizing subgraphs of G are computed, and the *symbol-size choice* phase in which the symbol

sizes of the nodes are actually chosen. Note that the preprocessing phase does not depend on K.

Phase 1. Preprocessing.

Fix a node v and an integer k corresponding to a legal symbol size at v. Let $S_v(k)$ be the set of all k-descendants of v. For the sake of uniformity of notation define $S_v(0) = S_v = \{v\}$. Let $G_{S_v(k)}$ denote the subgraph of G induced by the set $S_v(k)$ and by descendants of all elements of this set. Also as before, we use G_v to denote the subgraph of G induced by a node v and all of its descendants.

Let $n(G)$ denote the number of nodes in a graph G. Compute $n(G_{S_v(k)})$ for every node v of the graph G and every integer k corresponding to a legal symbol size at v.

Let

$$\alpha_v(k) \stackrel{\text{def}}{=} \frac{n(G_{S_v(k)})}{\sum_{u \in S_v(k)} n(G_u)}.$$

Clearly, $0 < \alpha_v(k) \leq 1$. Intuitively, $\alpha_v(k)$ measures how similar is the graph $G_{S_v(k)}$ to a tree. If $G_{S_v(k)}$ is a tree, then subgraphs G_u, $u \in S_v(k)$, are pairwise disjoint and hence $\alpha_v(k) = 1$. On the other hand, a small $\alpha_v(k)$ indicates that subgraphs G_u, $u \in S_v(k)$, have large intersections.

Compute $\alpha_v(k)$ for every node v and every integer k corresponding to a legal symbol size at v.

Phase 2. Symbol-size choice

This phase is based on the idea of the symbol size selection algorithm for trees, with a crucial change in the formula used to choose the symbol-size at each node.

For every node v of G, let L_v denotes the list of triples (d, pc_v^d, k), where d is a guaranteed surplus in v, pc_v^d is an estimation of the real cost associated with this surplus (will be defined later), and k is the legal symbol size at v that guarantees this surplus in v. Since a surplus d can be satisfied by any surplus $d' \geq d$, for any two entries $(d_1, pc_v^{d_1}, k_1)$ and $(d_2, pc_v^{d_2}, k_2)$ of L_v, where $d_1 \leq d_2$ and $pc_v^{d_1} \geq pc_v^{d_2}$, the first entry can be deleted as the second entry provides a better alternative. We will

implement this observation while constructing the list in the following.

For any leaf v, L_v is $\{(0,0,0)\}$. For every internal node v of G, assuming that the lists corresponding to all descendants of v have been computed, the following information must be computed for generating L_v:

- For every integer k corresponding to a legal symbol size at v, let $D_v(k)$ be the union of surpluses guaranteed by all k-descendants of v shifted by $k - K$:

$$\begin{aligned} D_v(k) \;=\; \{d + k - K \mid \forall u \in S_v(k), \exists (d', c_u^{d'}, k') \in L_u \text{ such that} \\ d' \geq d \text{ , and } d' = d \text{ for at least one } u \in S_v(k)\}. \end{aligned}$$

- The set of all possible surpluses at v, denoted by D_v is obtained as follows:

$$D_v = \bigcup \{D_v(k) \mid k \text{ is a legal symbol size at } v\}$$

- For every surplus $d \in D_v$, a graph G_v^d guaranteeing a surplus of d at v and equivalent to G_v is computed; a positive integer pc_v^d, called the *pseudo-cost* of G_v^d, and a corresponding legal symbol size are known (as explained in the following). The point (d, pc_v^d, k) must now be added to L_v.

- for any two entries $(d_1, pc_v^{d_1}, k_1)$ and $(d_2, pc_v^{d_2}, k_2)$ of L_v where $d_1 \leq d_2$ and $pc_v^{d_1} \geq pc_v^{d_2}$ the first entry is deleted from the list.

Computing G_v^d and pc_v^d

Fix symbol size k at node v. For every $d \in D_v(k)$ let $G_v^d(k)$ be the graph rooted at v and defined as follows:

there are $|S_v(k)|$ directed edges from v going to all nodes of $S_v(k)$ and $2^k - |S_v(k)|$ directed edges from v going to new leaves with label *reject*.

To each node $u \in S_v(k)$ attach the tree $G_u^{d+(K-k)}$ rooted at u, if existed, and $G_u^{d'+(K-k)}$, with smallest pseudo-cost among all the trees with $d' \geq d$, if otherwise.

The pseudo-cost of $G_v^d(k)$ (if this tree is defined) is:

$$pc(G_v^d(k)) = N_{TCAM}(v) + \alpha_v(k) \sum_{u \in S_v(k)} pc_u.$$

where:

$$pc_u = \begin{cases} pc(G_u^{d+(K-k)}) & \text{if } G_u^{d+(K-k)} \text{ exists} \\ pc(G_u^{d'+(K-k)}) & \text{Otherwise; } G_u^{d'+(K-k)} \text{ has the smallest} \\ & \text{cost among all the graphs with } d' \geq d \end{cases}$$

For a fixed d, let s_v^d be equal to the integer k for which the above pseudo-cost is minimum, where the minimum is taken over all k corresponding to legal symbol sizes at v. Define G_v^d to be $G_v^d(s_v(d))$ and pc_v^d to be the pseudo-cost of this graph. If $G_v^d(k)$ is not defined for any k then G_v^d is not defined.

Let r be the root of G. The graph $G_r^d, d \geq 0$, with minimum pc_r^d is the output of the algorithm. Now traversing the output tree top-down from r, symbol sizes at each node can be retrieved. If no graph $G_r^d, d \geq 0$, is defined, the problem does not have any solution for the given input G and K. □

Remark. The intuitive idea behind the definition of pseudo-cost is the following. Since the cost of $G_v^d(k)$ cannot be related in a simple way to the sum of costs of $G_u^{d'}$, taken over $u \in S_v(k)$, we make the simplifying assumption that the proportion between these numbers is the same as between the cost of $G_{S_v(k)}$ and the sum of costs of G_u, taken over $u \in S_v(k)$. This assumption, which holds for trees, is not true in general. Hence the resulting function, computed recursively, is not the cost but some distortion of it (hopefully relatively close to the actual cost), which we call

pseudo-cost. This new function is easy to compute, and we use it to make decisions concerning the choice of symbol-size. In each case the decision is determined by minimality of pseudo-cost, instead of cost.

Theorem 3.3.3. *The heuristic-based symbol size selection algorithm works in time $O(n^2)$ for an arbitrary input elementary graph of size n.*

Proof. Consider phase 1. For any node v and any integer k corresponding to a legal symbol size at v, the graph $G_{S_v(k)}$, all graphs G_u, for $u \in S_v(k)$, their costs, and consequently the coefficient $\alpha_v(k)$, can be computed in time $O(n)$. Hence phase 1 takes time $O(n^2)$. The same argument as for "symbol size selection algorithm for trees" shows that phase 2 takes time $O(nK)$. Hence the total execution time of is $O(n^2)$. □

3.3.4 Resolving Cycles of Cyclic Graphs

The dynamic programming techniques that were presented in this section are applicable to directed acyclic graphs. However, in most of the practical content inspection problems, the graph representation of policy rules contains cycles. For instance, directed graph representation of any content inspection processing that looks for specific words in the payload of incoming packets, will be a cyclic graph. In order to use the same dynamic programming techniques for optimizing TCAM requirements of such a content inspection problem, we resolve the cycles of the graph by decomposing it into a number of acyclic graphs. The set of acyclic graphs obtained from this decomposition represents the same policy rules that are represented by the initial cyclic graph. As such, the dynamic programming techniques can be applied to each acyclic graph of this set. The following example shows how a cyclic graph can be decomposed into a number of acyclic graphs.

Example 3.3.2. *Let us consider a simple content inspection problem in which all incoming packets are terminated with the string '000' and we want to check whether each incoming packet contains the string '110'. If there is a '110' in the packet, the packet is rejected and if not, the packet is accepted. We assume that the terminating string happens only once at the end of each packet. Figure 3.12-(a) represents the graph representation of this simple policy rule which is a cyclic graph. In order to decompose this graph into a number of acyclic graphs we first have to choose a limit, h, for the height of generated acyclic graphs. Let us choose* $h = 3$ *for this example. The first acyclic graph represents all the strings of length h or less which are obtained by walking on the cyclic graph, starting from the initial node* S_0*. This acyclic graph is represented in Figure 3.12-(b). The next acyclic graphs are obtained similarly, each one starting from a non-accepting node of the graph that was reached in a previous walk. For this cyclic graph, there are totally* 4 *more acyclic graphs, starting from states A, B, C, and D of Figure 3.12-(a). For instance, the two acyclic graphs that are generated by the walks starting from states A and B are shown in Figure 3.12-(c) and Figure 3.12-(d), respectively.* ◀

3.4 Speed Evaluation of Content Inspection Engines

The essential criterion on the performance of a CIE, is the speed of processing data packets; it is necessary that the solution provides a real-time performance. In other words, independent of the stream of incoming packets, the solution should always provide a processing speed as high as the incoming line-speed. Therefore, it is necessary to have a computable criterion on the speed performance of CIEs in order to determine the maximum line-speed in which the solution still provides a wire-speed

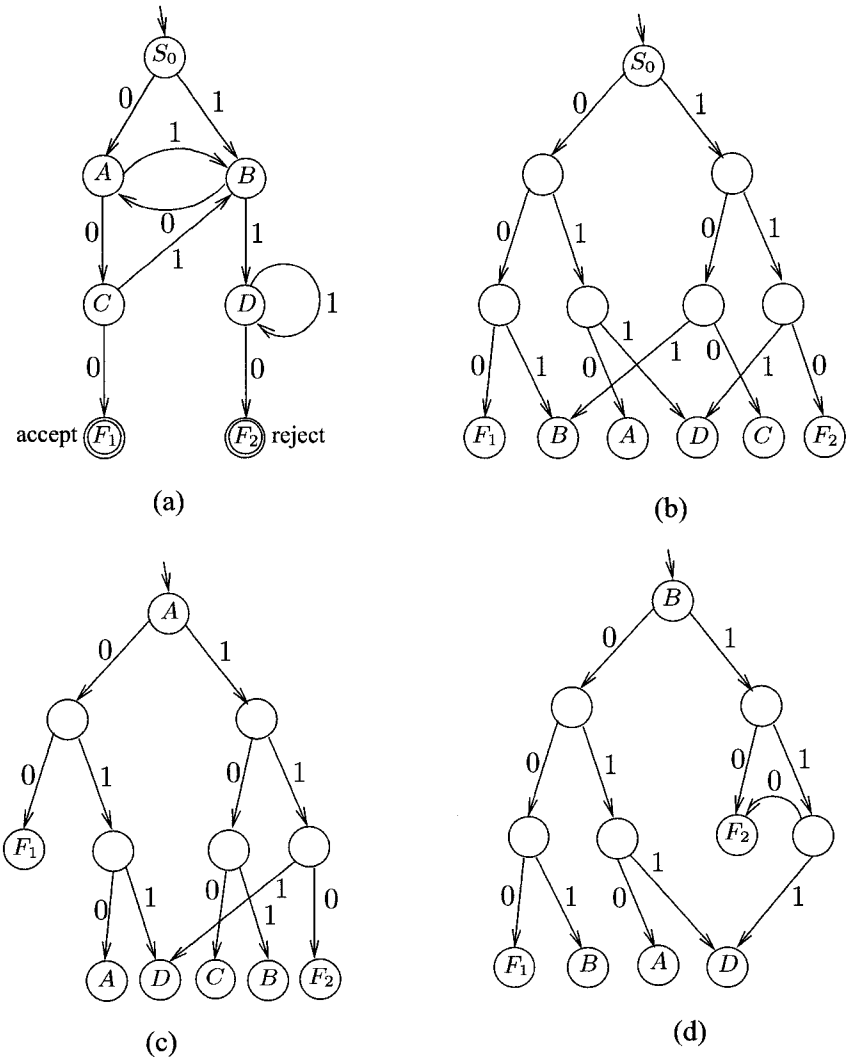


Figure 3.12: (a)- A graph representation of a sample policy rule and (b-d)- Three of the five acyclic graphs obtained from decomposing the cycles of this graph.

performance.

Traditional metrics for this wire-speed operation are the number of packets or bits processed per second. Such metrics of speed are appropriate when processing of each packet has constant (or at most well-bounded) per-packet component (e.g., multifield packet classification), or it has a flat per-byte cost (e.g., packet reception and storage, calculation of checksums). For example, in multifield packet classification, packets are analyzed and classified based only on some fields from the packet header, belonging to layers 3 and 4. As such, all packets receive the same amount of processing independent from their payload size. This characteristic allows us to quantify the speed of classification algorithms by calculating the number of packets which are processed in a time unit, assuming the worst-case scenario, i.e., considering the shortest length for all packets.

In the more general problem of content inspection, however, the amount of processing needed for each packet depends mainly not on the size of the payload, but on its contents. For example a malformed packet with a big payload size could be processed faster than a packet with a small size. This characteristic of the content processing prevents us to find the worst-case traffic, in terms of the required processing time, independent from the inspection algorithm.

We propose a criterion, called *worst-case throughput*, for evaluating the speed performance of an arbitrary CIE. This criterion is defined as the infimum of the ratio of the length of a processed input packet to its processing time, where the infimum is calculated over all possible input packets. Note that operation of the CIE under the worst-case throughput speed happens once the CIE receives a handcrafted input stream (from a malicious attacker) that keeps the system running at the lowest possible speed.

In order to prove the computability of this criterion, in the following, we will

explain a graph-based model which can be used to model the content inspection operation of every CIE (both software-based and hardware-based). Having such equivalent graph-based representation, we show how the worst-case throughput of a given solution can be expressed in terms of its equivalent graph parameters.

3.4.1 Graph-based Model of CIEs

Suppose that we have a software-based solution for a content inspection problem. In other words, we have a content inspection program which is supposed to run on a general-purpose CPU and perform content inspection on the incoming data stream. Such a program consists of a sequence of internally non-recursive decision blocks. Every block contains a number of possible cases, each case intended to inspect a segment, e.g., one byte, of the incoming packet. The packet segments will be consecutively processed in these decision blocks and based on their values, the control of the program is each time transferred from one block to another block. Since the packet can be very large in size, we assume that processed segments cannot be saved for future processing. As such, we assume that the packet is being sequentially processed in the blocks.

This sequence of decision blocks can be graphically represented by a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of edges of G. In this graph, each decision block of the inspection program is represented by a node $v \in V$, and corresponding to every possible case in the block, there is an outgoing edge from that node to an appropriate node of the graph. For example, suppose that b_1 and b_2 are two blocks of the inspection program with corresponding nodes v_1 and v_2, respectively. If there is a branching instruction inside block b_1, to block b_2 upon reading a segment l of the packet, then there would be an edge from v_1 to v_2, labeled by l, in G.

The program inspects the contents of the packet, starting from the first block,

where it scans a segment of data from this packet, and branches from the current block to another block according to the scanned segment. This procedure is repeated until it reaches a block where the program can make a decision, determining the result of the content inspection process for the packet. Every such block, that the program can terminate processing of a packet in it, is called a *stopping* block[3]. The node corresponding to the first block of the program is called the *start* node of G. We also add a *final* node to the directed graph and establish an unlabeled edge from every node corresponding to a stopping block into the final node. This extra node allows us to make a one-to-one correspondence between the inspection process of the packet and a walk on the directed graph from the start node to the final node; in every node of this path, the outgoing edge labeled with the scanned segment of data at the corresponding block is selected.

Note that the content inspection program has a limited number of decision blocks and those blocks may branch to each other recursively in order to inspect the contents of input packet with large sizes. A recursive branching of the decision blocks to each other translates to the existence of cycles on the corresponding directed graph.

The functionality of a hardware-based CIE is similar to that of a software-based CIE as it repeatedly scans a segment of the packet and changes its state according to the contents of the scanned segment. Therefore, we can make an argument similar to what was presented for the software-based CIEs, to show that this process can be modeled by a directed graph (with loops). In particular, for our high-performance FST based CIEs, this graph can be easily obtained from the state diagram of the high-performance FST, by adding a final node and its incoming edges to the state diagram. For example, consider the CIE based on the high-performance FST that was implemented by the TLT of Figure 3.5. The state diagram of the high-performance FST and the directed graph for modeling the functionality of the CIE are shown in

[3]Potentially, every block can be a stopping block.

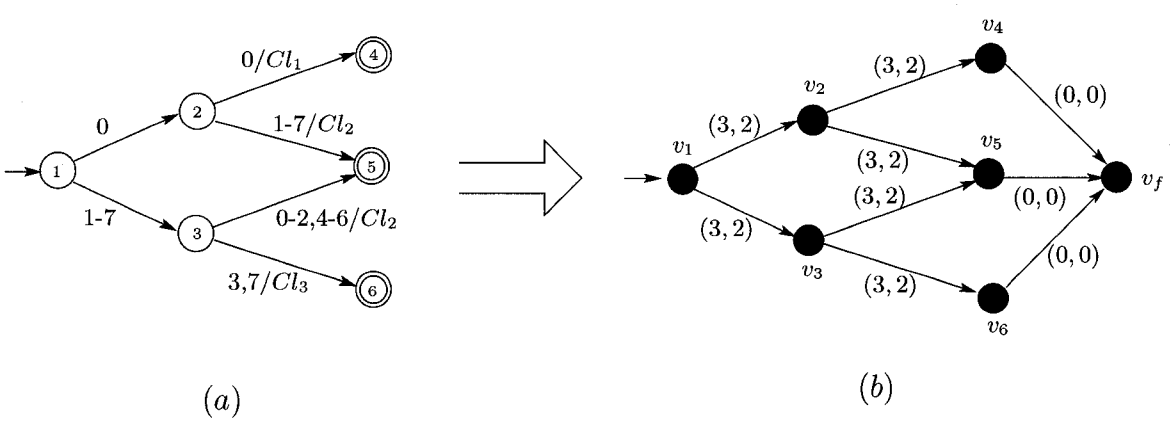


Figure 3.13: (a)- The state diagram of a high-performance FST (b)- The directed graph for modeling the CIE corresponding to this high-performance FST.

Figure 3.13.

So far, we have argued that independent from the architecture of a CIE, its operation can be modeled using a directed graph with loops. We now show how we can use this graph to calculate the worst-case throughput of the CIE. Every edge-label of the graph represents the bits which have been scanned while making this transition. We associate two other values to each edge: the first value is called the *weight* of the edge and is defined as the number of input bits in the label of the edge. The second value is called the *transition time* of the edge and is defined as the time it takes for the CIE to process the label of that edge. For example, if the CIE is a software-based CIE, the transition time of each edge is equal to the time that CPU needs to execute the case instructions corresponding to that edge. The weight and transition time of an unlabeled edge are defined to be zero. In Figure 3.13-(b), the pair of numbers on each edge show the weight and transition time associated with the edge; For all edges, except the edges incoming to the final node, the weight is 3 and transition time is 2 memory cycles.

A *path* on the directed graph is defined as a sequence of consecutive edges, with

repetitions if loops are involved, where the first edge is outgoing from the start node and the last edge is incoming into the final node. The weight and transition time of such a path, p, are defined as the summation of the weights and transition times of all edges in p, respectively. As it was mentioned earlier, every path p of the directed graph corresponds to the content inspection of an input packet. Therefore, if P denotes the set of all paths (possibly an infinite set), the worst-case throughput of CIE can be expressed as:

$$\textit{worst-case throughput} = \inf_{p \in P} \frac{w(p)}{t(p)}$$

in which, $w(p)$ and $t(p)$ denote the weight and transition time of p, respectively.

Example 3.4.1. *In order to clarify the correspondence between the functionality of a CIE and its graph-based model, we consider an abstract example of a single-rule content inspection problem. Although the amount of processing performed in this example is much less than what is required for a real-life content inspection problem, it shows the above-mentioned correspondence.*

Suppose that the incoming packets to the CIE have the structure illustrated in Figure 3.14. The HTTP-Payload of a packet may contain the word `COOKIE`*. Also the end of the HTTP-Payload is always specified by* `\n\n`*. The content inspection policy contains only one rule to reject every received packet unless it has a valid Source MAC Address, Destination Port Number = 80, and does not contain the word* `COOKIE` *in its HTTP-Payload.*

Now assume that we are provided with a software-based CIE which solves this content-inspection problem in 5 successive blocks as follows. The number of processed bits and also the running time for the different branching instructions of each block are written in parentheses (we suppose that the processing times of the branching instructions have been provided to us in terms of the number of memory accesses required for executing the branching instructions).

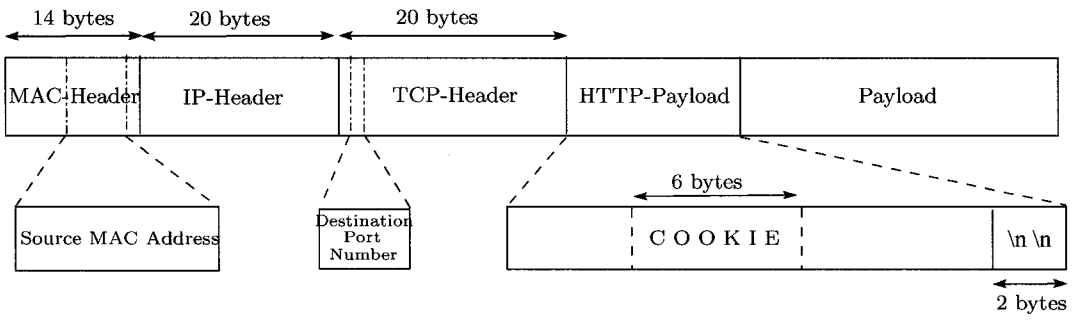


Figure 3.14: Format of the packets received by the CIE of Example 3.4.1.

Block 1. *Compare the Source MAC Address against the entries of a table containing invalid Source MAC Addresses. If there is a match, go to block 4. Otherwise go to block 2 (For both cases, number of processed bits = (14+20) × 8= 272, processing time = 10 memory access time).*

Block 2. *Check the Destination Port Number in the TCP-Header. If it is not equal to 80, go to block 4. Otherwise, set* `current position` *= 1 and go to block 3 (For both cases, number of processed bits = 20 × 8 = 160, processing time = 4 memory access time).*

Block 3. *Compare 6 bytes of the HTTP-Payload starting at* `current position` *against the word* `COOKIE` *and, simultaneously, 2 bytes of it, starting at* `current position` *against* `\n\n`. *If there is a match with* `COOKIE`*, go to block 4 (Number of processed bits is at least 6 × 8 = 48, processing time = 6 memory cycles). If there is a match with* `\n\n` *(i.e., we have reached to the end of the HTTP-Payload without finding the word* `COOKIE` *in it) go to block 5 (Number of processed bits = 2 × 8 = 16, processing time = 6 memory access time). Otherwise, set* `current position` *=* `current position` *+ 1 and go to the beginning of block 3 (Number of processed bits = 1 × 8 = 8, processing time = 6 memory access time).*

Block 4. *Reject the packet.*

Block 5. *Accept the packet.*

In order to obtain the worst-case throughput of this CIE, we model its operation with the graph illustrated in Figure 3.15. Each block of the CIE has a corresponding node in this graph; The nodes v_1, v_2, v_3, v_4, *and* v_5 *correspond to Blocks* $1, 2, 3, 4$, *and* 5, *respectively. The start node,* v_1, *is specified with an incoming arrow. The left and right numbers in the parenthesis of each labeled edge represent the number of processed bits and the transition time corresponding to that edge, respectively. Also,* v_f *is the final node of the graph and there is an unlabeled edge from every node corresponding to a stopping block to this node. Note that when the packet does not have proper TCP-Header or HTTP-Payload (for example, a packet with Destination Port Number= 80 which does not have any HTTP-Payload), blocks* 2 *or* 3 *would be stopping blocks, respectively. We have assumed that the packets that are less than 34 bytes in size are deleted before arriving to the CIE, i.e., all the incoming packets to the CIE have complete MAC-Header and IP-Header.*

For this example, the worst-case throughput of the CIE is 8 bits per 6 memory cycles; For every $\epsilon > 0$ *there is a path* p_ϵ *on the graph, going through the only loop of the graph enough times, for which we have* $\frac{8}{6} < \frac{w(p_\epsilon)}{t(p_\epsilon)} \leq \frac{8}{6} + \epsilon$. *Nonetheless, there is no finite path on the graph whose weight to time ratio is equal to the worst-case throughput of 8/6 (bits/memory cycle). This is, in fact, why the worst-case throughput of a CIE should be defined as the infimum (and not as the minimum which may not exist) weight to time ratio of all paths on the graph.*

Assuming a memory access time of 5 nsec, the worst-case throughput is equal to 266 Mbits/sec, meaning that this CIE cannot provide a wire-speed performance for an input line with a speed higher than 266 Mbits/sec. In order to improve the worst-case throughput, one should improve on the algorithm that searches for the floating pattern

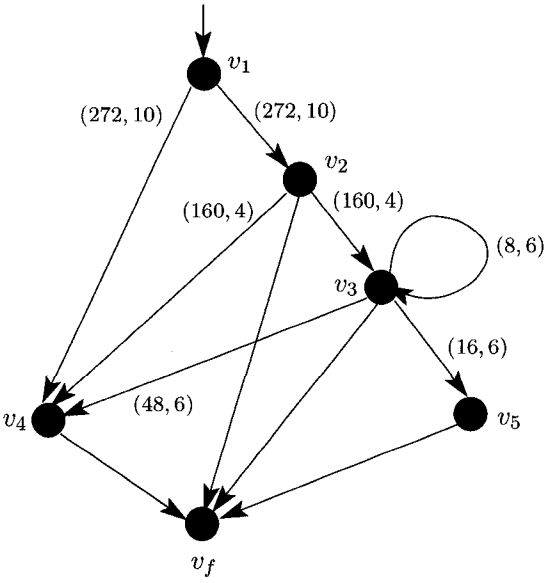


Figure 3.15: Graph-based model of the CIE of Example 3.4.1.

(`COOKIE` or `\n\n`) in the HTTP-Payload. ◀

3.4.2 Calculation of the Worst-Case Throughput

In a real-life content inspection problem, the graph-based model of the CIE normally has thousands of nodes and edges, and it would not be possible to find the worst-case throughput without having an efficient algorithm for this job. We show that the problem of calculating the worst-case throughput of a CIE can be transformed into an MWTR problem. The MWTR problem has numerous applications in graph theory [49], discrete event system theory [50], and performance analysis of synchronous and asynchronous digital systems [51, 52]. As a result, many algorithms have been proposed for solving this problem [42, 45, 49, 51, 53].

The MWTR problem can be expressed as follows. Suppose that $G = (V, E)$ is a directed graph in which every edge e in E is associated with two numbers: a weight, $w(e)$, and a transition time, $t(e)$. The weight and transition time of a cycle C in G are defined as the sum of the weights and transition times of the edges on C, respectively.

The *weight to time ratio* of C is defined as,

$$\rho(C) = \frac{w(C)}{t(C)},\ t(C) \geq 0$$

The *minimum weight to time ratio* of G, $\rho^*(G)$, is defined as:

$$\rho^*(G) = \min_{C \in G} \rho(C)$$

Now consider a CIE with its equivalent graphical representation $G = (V, E)$. In order to transform the worst-case throughput calculation problem into the MWTR problem, we establish an unlabeled edge outgoing from the final node of G into its start node and define the weight and the transition time of this edge to be equal to zero. It can be easily verified that the worst-case throughput of the CIE is equal to the MWTR of the new graph.

There are many efficient algorithms for solving the MWTR problem. For our experimental works, we have used the Lawler's algorithm of [45], which can be explained as follows. Let G_λ denote the graph obtained from G by subtracting λ from the weight to time ratio of every edge. The MWTR, λ^*, is the largest λ such that G_λ does not have any cycle with negative weight to time ratio. Suppose that *min* and *max* are the minimum and maximum weight to time ratio of the edges in G, respectively. Using the fact that $\lambda^* \in [min\ max]$, Lawler's algorithm performs a binary search on the range $[min\ max]$ for finding an ϵ-approximation of λ^*. In each step of the binary search, it considers a value from this range as a possible candidate for λ^* and runs the Bellman-Ford algorithm to check for a negative cycle in G_λ. If a negative cycle is found, then $\lambda < \lambda^*$ and thus λ is increased. Otherwise, λ is decreased. The algorithm terminates when the interval for possible values of λ^* becomes smaller than ϵ.

3.5 Simulation Results

In order to evaluate the performance of a CIE based on a high-performance FST generated by TLT architecture, in this section we consider a real life content inspection problem. The set of policy rules of this problem has 105 network intrusion detection rules taken from the Snort database [54]. The reason for having 105 rules in the database is to compare the performance of the CIE based on our architecture with that of [20] which also implements a CIE with 105 intrusion detection rules. The solution presented in [20], is one of the few hardware-based proposed solutions we have found that implements both the header and the payload parts of the rules and also is accompanied with exact performance results.

3.5.1 CIE Generation Procedure

The CIE generation procedure takes the set of inspection policy rules, together with a given operation speed for the CIE, as the input and it goes into the following three steps to generate an efficiently implementable high-performance FST for this problem.

First Step

The first step is to determine how many bits should be processed, on average, at each state of the high-performance FST. It was explained earlier that for a high-performance FST implemented in TLT, each state transition takes one state memory access time plus one layout processing time. For this example we consider the same line speed of [20] which is 2.88 Gbits/sec. Also we suppose that the state memory nodes and layout entries are implemented in SRAM and TCAM, respectively. With today's available technology, an access time of 5.5 nsec is a reasonable assumption for both SRAM and TCAM. This means that each state transition takes 11 nsec and thus at least 32 bits should be processed on average, at each state of the high-performance

FST.

Second Step

An average number of at least 32 processed bits in the states of the high-performance FST can be met with many different high-performance FSTs with different memory requirements. We have developed a TLT generation program which uses the dynamic programming techniques of Section 3.3 to find one of those, with a low memory requirement. The rules of the incoming policy rule-set are first merged to produce a one-bit transition directed graph (i.e., an elementary graph) similar to Figure 3.2. This graph is used as the input to the TLT generation program. The TLT generation program first resolves the cycles of this graph by breaking it down to a number of acyclic graphs. Then it applies the dynamic programming techniques of Section 3.3 to these acyclic graphs to determine the number of bits which have to be processed in each state of the high-performance FSTs. Finally it implements the states of the high-performance FST in state memory nodes and encodes the corresponding layout entries in TCAM[4].

Third Step

The TLT generation program is guaranteed to generate the TLT implementation of a high-performance FST with an average number of at least 32 bits processed at each state. This means that the worst-case throughput of the CIE based on this high-performance FST will be greater than or equal to 2.88 Gbits/sec (assuming a value of 5.5 nsec for SRAM and TCAM access times). In order to obtain the exact value for the worst-case throughput, we apply the Lawler's algorithm to the graph based model of the CIE. Our implementation of the algorithm gets the TLT implementation

[4]Note that the output of the TLT generation program is just a local optimum solution to this problem based on the parameters that are chosen and given to the TLT generation program.

of the high-performance FST from the TLT generation program, generates the graph based model of the CIE, and then applies the Lawler's algorithm to it. The exact value of the worst-case throughput for this CIE is 57.7 bits per state transition time (11 nsec), or 5.2 Gbits per second.

3.5.2 Evaluating the Simulation Results

In order to compare our solution with that of [20], we considered 105 intrusion detection rules and operation speed of 2.88 Gbits/sec for the CIE. Combining these 105 rules generated a one-bit transition directed graph with 22780 nodes. Then the TLT generation program used this graph to generate an equivalent high-performance FST with an average number of more than 32 processed bits at each state. The TLT implementation of this high-performance FST has only 15964 state memory words which can be implemented in an SRAM with a size less than 0.5 Mbytes. Before finding identical layouts, these state memory words refer to 966 layout entries for layout processing, which are implemented by 34232 TCAM rules. After finding identical layout entries, the number of layout entries and required TCAM rules reduce to 139 and 7455, respectively. This means a 78 percent reduction in TCAM size. The maximum length of the generated TCAM rules is 64 bits and thus the TCAM rules can be implemented in a TCAM of size 0.5 Mbits.

While we started the design of this CIE with a target speed of 2.88 Gbits/sec, the output from the third step of the memory optimization procedure shows that the exact value for the worst-case throughput is 5.2 Gbits/sec.[5] This is 1.7 times faster than the 2.88 Gbits/sec CIE of [20] in which each packet is compared against all the 105 rules in parallel and needs more expensive hardware: at least one Altera EP20K series FPGA chip.

[5]Parallel processing of two packets in this CIE (by adding a second current state register to the CIE as it was explained in Section 3.1) doubles the worst-case throughput to 10.4 Gbits/sec.

Note: The considerable difference between the target speed of 2.88 Gbits/sec and the worst-case throughput of the generated CIE is an indication that the heuristics used in the TLT generation program have minimized the required memory by choosing an average number of processed bits (at each state) considerably larger than 32. i.e., 57.7.

3.6 Conclusions

A wire-speed solution for the problem of content inspection should tackle the complex payload processing and high memory requirements of this problem. By processing a large enough number of bits at each of its states, a high-performance FST has the potential to provide a wire-speed solution to this problem. We presented the TLT architecture for implementing a high-performance FST which avoids a large memory increase with the number of rules. We also presented some dynamic programming techniques for generating TLTs with reduced TCAM requirements. Our simulation results for a given content inspection problem confirm that the TLT architecture implements high-performance FSTs efficiently.

Due to the inclusion of payload processing in content inspection, traditional speed evaluation metrics, such as number of packets processed per second, cannot be used for speed evaluation of CIEs. We introduced *worst-case throughput* as an appropriate criterion for the speed evaluation of CIEs and presented a graphical model which makes its calculation possible.

A challenging stage in the process of generating the TLT corresponding to a high-performance FST is to efficiently represent the incoming string labels on the outgoing edges of each state in TCAM. In general case, if n bits are processed in a given state, the labels on a single outgoing edge of that state are a subset of the strings of length n over the binary alphabet. In this chapter we proved that this problem is at least, as

difficult as the NP-complete problem of finding a minimal Disjunctive Normal Form of a logical statement [41]. However, if the subset of strings of length n happens to be a consecutive set of binary numbers, then the problem may be easier to solve. This special case is, in fact, the problem of range-matching in TCAM which is faced in the today's industrial solutions for multifield packet classification. We will investigate this problem in Chapter 5.

Chapter 4

Packet Processing by Concatenation Transducers

FSMs can express the same family of languages that regular expressions describe: regular languages [35]. As such a content inspection problem that processes strings of a non-regular language, cannot be expressed and solved by a high-performance FST. An important example of such content inspection problems is XML processing whose underlying language, XML, is a non-regular language. Concatenation state machines of [55, 56] introduce a stack to FSMs to extend their expressive power to the family of *simple languages* [57, 58] which include a large class of the non-regular languages that have a non-trivial nesting structure. For example simplified XML is a simple language.

Addition of an output tape, for generating output strings, to a concatenation state machine enables it to process packets that have content strings belonging to simple languages. We call such a transducer a *concatenation transducer (CT)*. In this chapter we present CTs and then show that they can be implemented by adding a stack to the TLT architecture. As calculating the worst case throughput of CTs cannot be done by available algorithms, we also present a new algorithm for solving this problem, together with the proof of its correctness and complexity evaluation.

The outline of this chapter is as follows. Section 4.1 presents formal definition of

a CT and explains its operation and implementation by TLT architecture. In Section 4.2 we introduce our algorithm for the worst-case throughput calculation of CTs. The correctness of the algorithm and its complexity is presented in Section 4.3. Finally, the chapter is concluded in Section 4.4.

4.1 Concatenation Transducers

In this section, we present the formal definition of CTs as a directed graph. This directed graph is used in the next section to express and calculate the worst-case throughput of CTs. We also show that CTs can be efficiently implemented by including a stack in the TLT architecture.

Definition 4.1.1. A CT is a rooted directed graph with possibility of multiple edges, which can be characterized by an 8-tuple $M = (\Sigma, \Omega, S, C, A, \eta_S, \eta_C, v_{in})$, where:

- Σ, Ω, S, C, A are finite sets of *input symbols*, *output symbols*, *switch nodes*, *concatenation nodes*, and *accepting nodes*, respectively;
- $\eta_S : (S, \Sigma^*) \mapsto (S \cup C \cup A, \Omega^*)$ is a partial mapping determining the destination nodes and the output strings corresponding to the labeled outgoing edges of the switch nodes;
- $\eta_C : (C, \{\texttt{left}, \texttt{right}\}) \mapsto (S \cup C \cup A)$ is a partial mapping determining the destination nodes corresponding to the outgoing edges of the concatenation nodes;
- $v_{in} \in (S \cup C \cup A)$ is the *initial node.*

It is required that η_S be *prefix deterministic*, i.e., for every switch node $s \in S$ and input words $u, w \in \Sigma^*$, if $\eta_S(s, u)$ and $\eta_S(s, w)$ are both defined, then neither u nor w is a proper prefix of the other.

This definition of CTs is slightly different from the original definitions of [55]. Namely, we consider the high-performance version of a CT, in which an outgoing edge of a switch node is labeled by an input string instead of a single input symbol.

Figure 4.1 shows an example of a CT. It is built over input alphabet $\Sigma = \{a, b, c\}$ and output alphabet $\Omega = \{X, Y, Z\}$. The *initial node* of the CT, v_{in}, is distinguished by a short incoming arrow. The only switch node of this CT, s_1, is represented by a plain circle while the concatenation nodes v_{in}, c_1 and c_2, are represented by rectangles with `left` edge outgoing from the left and `right` edge outgoing from the right part of the node. The outgoing edges of concatenation nodes are shown using dash lines. Accepting nodes are represented by double circles.

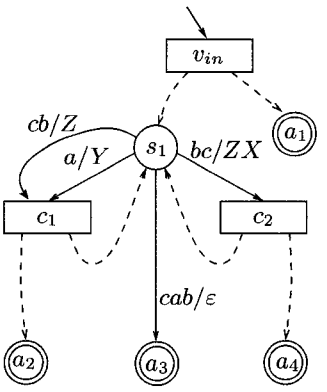


Figure 4.1: A Concatenation Transducer.

A CT $M = (\Sigma, \Omega, S, C, A, \eta_S, \eta_C, v_{in})$ implements a partial mapping from words over the input alphabet Σ into words of the output alphabet Ω. This partial mapping can be computed by associating an *execution stack*, denoted by `stack` and represented as a list of nodes, to M. At the beginning of the execution process the execution stack contains one element, the initial node, i.e., `stack` $= (v_{in})$. Running the following **execution loop**, the CT processes the nodes of the execution stack until the stack becomes empty.

execution loop:

- The top node of the stack, t, is popped. Then,
  - If t is a switch node, $t \in S$, the transducer reads an initial segment of the input stream corresponding to a string $u \in \Sigma^*$ for which $\eta_S(t, u)$ is defined. It then pushes the node specified by $\eta_S(t, u)$ on the top of the execution stack and sends the string specified by $\eta_S(t, u)$ to the output. Here we say that the CT *goes through* the outgoing edge of the node t which is labeled by u, i.e., the edge that connects t to the node specified by $\eta_S(t, u)$. Note that, because of the *prefix deterministic* property of η_S, the initial segment of the input stream corresponding to u is unique.
  - If t is a concatenation node, $t \in C$, the machine pushes two nodes on the stack, first $\eta_C(t, \texttt{right})$ and then $\eta_C(t, \texttt{left})$ (so node $\eta_C(t, \texttt{left})$ is on the top of the execution stack). In this case, we say that the CT goes through the `left` edge right away and goes through the `right` edge later when the node $\eta_C(t, \texttt{right})$ is popped from the stack.
  - If t is an accepting node, $t \in A$, then the machine goes to the beginning of the next step of the loop without doing anything.
- If the execution stack is not empty the execution loop is continued. If the stack is empty, which is possible only if the last node of the stack was an accepting node, the machine performs the action corresponding to the last accepting node.

Intuitively, a switch node can be seen as a state in a normal transducer. A concatenation node, on the other hand, represents a "subroutine call" for the destination of the `left` edge; once the subroutine terminates, we continue with the destination of the `right` edge. The subroutines are terminated by accepting nodes that represent "return" instructions.

We say that M accepts w if starting with `stack` $= (v_{in})$ and reading the input word w, the execution stack becomes empty. The result is the concatenation of all outputs produced during processing w. The existence of concatenation nodes in CTs enables them to process strings of languages that cannot be processed by normal transducers; It has been shown in [56] that CTs accept the strings of simple languages which include the simplified XML. This inclusion implies that a content inspection problem which deals with recognizing and validating simplified XML packets can be solved by a CIE that is build as a CT.

4.1.1 Implementing Concatenation Transducers by TLT Architecture

CTs have already been implemented in the *PAX.port* engines of IDT Canada [48]. However, the one-level memory architecture of *PAX.port* engines limits their processing speed for large sets of inspection policy rules. The TLT architecture of Chapter 3 can be extended to efficiently implement a CT-based CIE; Figure 4.2 shows the components of the extended architecture. In comparison with the components of the TLT architecture of Figure 3.6, the only extra component that is required is a stack for implementing the functionality of concatenation nodes.

During the inspection process, the current state register of Figure 4.2 contains the information corresponding to the current state of the CT. This state can be a switch node, a concatenation node, or an accepting node. If the current state is a switch node, the current state register contains the number of packet bits that are processed at this switch node and the address of the TCAM corresponding to the switch node; The CIE reads the given number of bits from the input packet and sends them to the TCAM to be matched against the corresponding TCAM entry. The TCAM returns an offset which is used to determine the address of the next state memory entry that

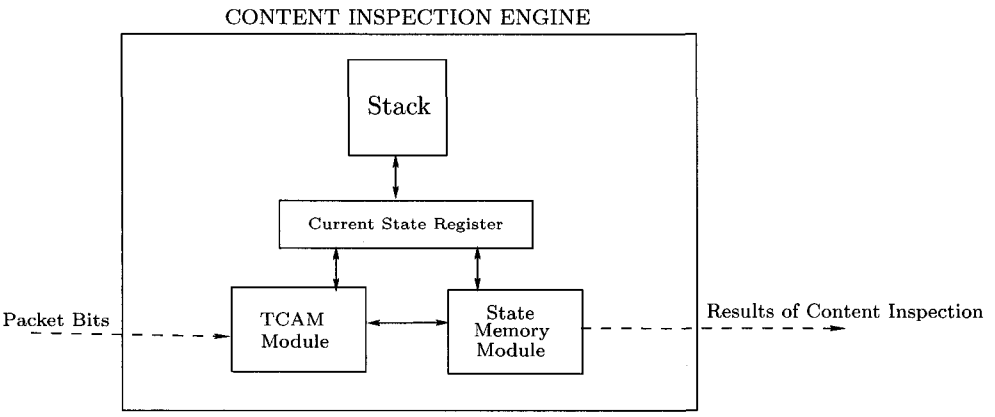


Figure 4.2: Structure of a CIE based on the TLT architecture.

should be loaded into the register[1].

If the current state is a concatenation node, then the current state register contains the addresses of the two nodes $\eta_C(t, \texttt{left})$ and $\eta_C(t, \texttt{right})$. In this case, first the address of the $\eta_C(t, \texttt{right})$ is pushed into stack and then the state memory entry at address $\eta_C(t, \texttt{left})$ is loaded as the into the current state register.

Finally, if the current state is an accepting node then a pop operation is performed on the stack. If the stack is not empty, then the state memory entry at the popped address is loaded into the current state register. If the stack is empty then action corresponding to the current state is performed.

Using this architecture for implementing CT-based CIEs and taking the conservative assumption that each of the push and pop operations take half the time of a memory access, the CIE performs two memory access in the concatenation nodes. In switch nodes, the CIE performs one TCAM access and one state memory access.

[1]Note that this "direct" loading of the next state memory entry from the state memory module is equivalent to the combination of the two consecutive stack-push and stack-pop operations in the execution loop: the push operation at the end of processing a switch node and its following pop operation at the beginning of next step of the execution loop.

4.2 The Worst-Case Throughput of Concatenation Transducers

In Chapter 3 we showed how we can calculate the worst-case throughput of a CIE using its graph-based model. The definition of a CT directly provides us with such a graph-based model. Nonetheless, due to the existence of concatenation nodes, this graph may be infinite and thus the worst-case throughput of CTs cannot be calculated by available algorithms on graphs. In this section we first give the formal definitions related to the worst-case throughput of a CT and then present our algorithm for calculating the worst-case throughput.

In a CT, a *path* p starting from a node u is defined as a finite sequence of edges, possibly with repetitions, which the CT can go through successively, starting with $\texttt{stack} = (u)$, to the end of the execution loop (i.e., ending with an empty execution stack). A path which corresponds to an accepted word (i.e., a path starting with $\texttt{stack} = (v_{in})$) is called an *accepting path.* Thus, an accepting path begins from an edge outgoing from the initial node and ends with an edge incoming to an accepting node. The number of edges in a path p is called *length* of p and is denoted by $|p|$.

For example, for the CT of Figure 4.1, the input string *abcbccab* is accepted producing output $YZXZX$, and the accepting path, p, is given by the following sequence of edges: $(v_{in}, \texttt{left}, s_1)$ (s_1, a, c_1) $(c_1, \texttt{left}, a_2)$ $(c_1, \texttt{right}, s_1)$ (s_1, bc, c_2) $(c_2, \texttt{left}, s_1)$ (s_1, bc, c_2) $(c_2, \texttt{left}, s_1)$ (s_1, cab, a_3) $(c_2, \texttt{right}, a_4)$ $(c_2, \texttt{right}, a_4)$ $(v_{in}, \texttt{right}, a_1)$. A triple (u, l, v) stands for an edge connecting node u to node v and its label l; If u is a switch node then l is an input string and if u is a concatenation node, $l \in \{\texttt{left}, \texttt{right}\}$. For the above example of p, we have $|p| = 12$.

We suppose that every node of a CT is "useful", i.e., there is at least one accepting path of the CT that goes through that node.

We define *weight function* w on the set of edges of a CT M into integer numbers

as follows:

- for every edge e outgoing from a concatenation node, $w(e) = 0$, and,
- for every edge e outgoing from a switch node, $w(e)$ is equal to the length of the input string label of that edge.

The weight function w on edges extends naturally to paths: $w(e_1 e_2 \dots e_k) \stackrel{\text{def}}{=} w(e_1) + w(e_2) + \dots + w(e_k)$. Notice that the weight of an empty path is 0.

The worst-case throughout of a CIE is the infimum of the ratio of the length of processed packets over their processing time. In a CT, each processed packet has a corresponding accepting path and thus the set of all accepting paths of the CT corresponds with the set of all processed packets. Besides, since the CIE performs two memory accesses in each node, the processing time of every processed packet is proportional with the number of edges in its corresponding accepting path.

Definition 4.2.1. The *throughput* $t(p)$ of a non-empty path p is defined as the weight of the path divided by its length. The worst-case throughput of a CT M, denoted by $wct(M)$, is equal to the infimum of the throughput of all non-empty accepting paths of the CT and can be expressed as:

$$wct(M) \stackrel{\text{def}}{=} \inf_{p \in P} \frac{w(p)}{|p|}$$

where P is the set of all accepting paths of M.

We have assumed without loss of generality that only the lengths of the input strings on the edges of a CT affect the worst-case throughput of the CT and the worst-case throughput is independent of the output strings; The algorithm that will be presented in this section for calculating the throughput can be easily extended to the cases that generating output strings cannot be performed in parallel with the reading of input strings.

As an example consider the path made by the following sequence of edges on the CT of Figure 4.1: (s_1, cb, c_1) $(c_1, \texttt{left}, a_2)$ $(c_1, \texttt{right}, s_1)$ (s_1, a, c_1) $(c_1, \texttt{left}, a_2)$ $(c_1, \texttt{right}, s_1)$ (s_1, bc, c_2) $(c_2, \texttt{left}, s_1)$ (s_1, cab, a_3) $(c_2, \texttt{right}, a_4)$. The number of edges on this path is 10 and the length of the string obtained from concatenating its input labels equals 8. Hence, the throughput of this path is $\frac{4}{5}$. As another example consider again the accepting path corresponding to string *abcbccab* which was given in previous section: $(v_{in}, \texttt{left}, s_1)$ (s_1, a, c_1) $(c_1, \texttt{left}, a_2)$ $(c_1, \texttt{right}, s_1)$ (s_1, bc, c_2) $(c_2, \texttt{left}, s_1)$ (s_1, bc, c_2) $(c_2, \texttt{left}, s_1)$ (s_1, cab, a_3) $(c_2, \texttt{right}, a_4)$ $(c_2, \texttt{right}, a_4)$ $(v_{in}, \texttt{right}, a_1)$. This path contains 12 edges, and accepts a word of length 8, hence its throughput equals $\frac{2}{3}$.

The worst-case throughput of the CT of Figure 4.1 is $\frac{1}{3}$. There is no accepting path on the CT which can achieve this throughput. However, for every $\delta > 0$, there is an $n \geq 1$ and an accepting path of the CT corresponding to the input string $a^n cab$, which has a throughput of $\frac{n+3}{3n+1} < \frac{1}{3} + \delta$.

4.2.1 An Algorithm for Worst-Case Throughput Calculation of CTs

The first step in presenting our algorithm for calculating an ϵ-approximation of the worst-case throughput of a CT is the following observation.

Lemma 4.2.1. *The worst-case throughput of a CT lies between the minimum and the maximum of the weights of its edges.*

Now suppose that we have an oracle, called `Lower_Bound`, which takes as input a CT, M, together with a real value μ, and returns `true` if μ is a lower bound for the worst-case throughput of M, i.e., $\mu \leq wct(M)$; or `false`, otherwise. Having the `Lower_Bound` oracle and using Lemma 4.2.1, we can design an algorithm for finding an ϵ-approximation of the worst-case throughput of a CT, M, where ϵ is a

positive real number; Let *min* and *max* be the minimum and maximum weights of the edges of M. The algorithm performs a binary search on the range $[min, max]$. In each step of the binary search, the algorithm considers a value μ from this range as a possible candidate for the worst-case throughput of M. It then applies the `Lower_Bound` oracle to verify if μ is a lower bound for the worst-case throughput, and halves the range accordingly. The algorithm continues as long as the approximation distance ϵ remains smaller than the length of the range. This algorithm, that we call `WC_Throughput_Approximation`(M, ϵ), is presented in Figure 4.3.

```
WC_Throughput_Approximation(M, ε)
    upper_bound := max
    lower_bound := min
    μ := (upper_bound + lower_bound)/2
    while (upper_bound − lower_bound ≥ ε) do
        if Lower_Bound(M, μ) then
            lower_bound := μ  else  upper_bound := μ
        μ := (upper_bound + lower_bound)/2
    return μ
```

Figure 4.3: `WC_Throughput_Approximation`(M, ϵ): finds an ϵ-approximation of the worst-case throughput of M.

Definition 4.2.2. For a CT M and an arbitrary real value μ we define *μ-updated weight function* w_μ on the set of edges of M into real numbers:

$$w_\mu(e) \stackrel{\text{def}}{=} w(e) - \mu$$

for every edge e of the CT.

The *μ-updated weight* of a path $p = e_1 \dots e_k$ is $w_\mu(p) \stackrel{\text{def}}{=} w_\mu(e_1) + \dots + w_\mu(e_k)$, and thus $w_\mu(p) = w(p) - |p|\mu$.

Our implementation of the `Lower_Bound` oracle is based on the following lemma.

Lemma 4.2.2. *For a CT M and a real value μ we have:*

$$\mu \leq wct(M) \Longleftrightarrow w_\mu(p) \geq 0, \textit{for every accepting path } p \textit{ of } M;$$

Proof. Let $\mu^* = wct(M)$:

$$\mu \leq \mu^* \Leftrightarrow \forall p \in P, \frac{w(p)}{|p|} \geq \mu^* \geq \mu \Leftrightarrow \forall p \in P, \frac{w_\mu(p)}{|p|} = \frac{w(p) - |p|\mu}{|p|} \geq \mu^* - \mu \geq 0$$

□

Lemma 4.2.2 states that, in order to verify if μ is a lower bound of the worst-case throughput of M, it is sufficient to check if every accepting path p satisfies $w_\mu(p) \geq 0$. Intuitively, our implementation of the `Lower_Bound` oracle is an extension of the Bellman-Ford single-source shortest path algorithm to the CTs with μ-updated weight function. If the shortest path (with respect to w_μ) exists, we simply check if its μ-updated weight is non-negative. If no shortest path can be found, then it means that there is an infinite number of paths with negative μ-updated weights. The `Lower_Bound` algorithm is presented in Figure 4.4.

4.3 Correctness and Complexity of the Worst-Case Throughput Calculation Algorithm

We have to prove that the `WC_Throughput_Approximation`(M, ϵ) algorithm returns a value which is an ϵ-approximation of the worst-case throughput of the CT M. Based on the arguments of the previous section, we only need to show the correctness of our implementation of the `Lower_Bound`(M, μ) oracle. For this purpose, we introduce the concept of *c-path.*

```
    Lower_Bound(M, μ)
1       for each v ∈ S ∪ C do  d(v) := ∞
2       for each v ∈ A do  d(v) := 0
3       j := |S ∪ C| + 1; change := true
4       while (change and j ≥ 1) do
5           change := false
6           for each s ∈ S do
7               tmp := min{w_μ(e) + d(x) | e = (s, l, x) is an edge in M}
8               if tmp < d(s) then d(s) := tmp; change := true
9           for each c ∈ C do
10              tmp := ∑{w_μ(e) + d(x) | e = (c, l, x) is an edge in M}
11              if tmp < d(c) then d(c) := tmp; change := true
12          j := j − 1
13      if (change or d(v_in) < 0) then return false else return true
```

Figure 4.4: `Lower_Bound`(M, μ): checks if μ is a lower bound for the worst-case throughput of M.

Definition 4.3.1. A *c-path* of a CT M is a pair (π, ϕ), where π is an ordered rooted tree and $\phi : \pi \mapsto M$ is a graph morphism such that, for every vertex x in π:

- if $\phi(x)$ is a switch node in M then x has exactly one outgoing edge.
- if $\phi(x)$ is a concatenation node of M then x has a pair (e_1, e_2) of outgoing edges such that $\phi(e_1) = (\phi(x), \texttt{left}, v)$ and $\phi(e_2) = (\phi(x), \texttt{right}, v')$, for some nodes v, v' in M.

The node $u = \phi(r)$, where r is the root of π, is called the *starting node* of the c-path. In order to underline that u is the starting node of the c-path, we will denote the c-path by (π_u, ϕ).

Note that the definition of c-path implies that for every leaf vertex x of the c-path (π_u, ϕ), we have $\phi(x) \in A$. Also every c-path corresponds to a path of the CT, and if u is the initial node, then the c-path corresponds to an accepting path of the CT. In fact, there is a one-to-one correspondence between the paths of a CT starting in a node v and the c-paths (π_v, ϕ): the (ordered) DFS traversal of each such c-path defines, through ϕ, a unique path of the CT starting with $\texttt{stack} = (v)$.

An example of a c-path corresponding to the accepting path $(v_{in}, \texttt{left}, s_1)$ (s_1, bc, c_2) $(c_2, \texttt{left}, s_1)$ (s_1, cab, a_3) $(c_2, \texttt{right}, a_4)$ $(v_{in}, \texttt{right}, a_1)$ of the CT of Figure 4.1, is illustrated in Figure 4.5.

Definition 4.3.2. Let $M = (\Sigma, \Omega, S, C, A, \eta_S, \eta_C, v_{in})$ be a CT, (π_u, ϕ) a c-path of M, and E the set of edges in π_u. We define:

- the *length* of the c-path: $|(\pi_u, \phi)| \stackrel{\text{def}}{=} |E|$, i.e., number of edges in π_u;
- the *weight* of the c-path: $w(\pi_u, \phi) \stackrel{\text{def}}{=} \sum_{e \in E} w(\phi(e))$;
- the *throughput* of the c-path: $\mathit{throughput}(\pi_u, \phi) \stackrel{\text{def}}{=} w(\pi_u, \phi)/|(\pi_u, \phi)|$;

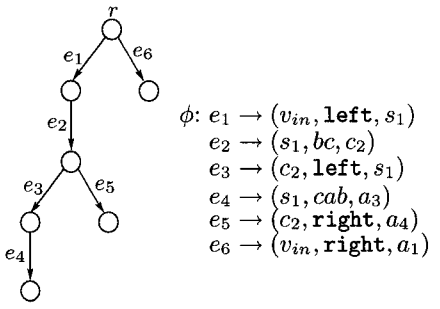


Figure 4.5: A c-path $(\pi_{v_{in}}, \phi)$ of the CT of Figure 4.1.

- the *μ-updated weight* of the c-path: $w_\mu(\pi_u, \phi) \stackrel{\text{def}}{=} \sum_{e \in E} w_\mu(\phi(e))$; and
- the *height* of the c-path, denoted by $h(\pi_u, \phi)$, as the height of the tree π_u.

It is easy to see that the length, weight, throughput, and μ-updated weight of a c-path are equal to those of its corresponding path in the CT.

Lemma 4.3.1. *Let u be a node of a CT M and $d_k(u)$ be the value of $d(u)$ at the end of the k-th iteration of the while loop of* `Lower_Bound`(M, μ). *We have:*

$$\forall(\pi_u, \phi),\ \textit{if } h(\pi_u, \phi) \leq k \textit{ then } d_k(u) \leq w_\mu(\pi_u, \phi) \tag{4.3.1}$$

and

$$d_k(u) = \infty \textit{ or } \exists(\pi'_u, \phi'),\ \textit{such that } w_\mu(\pi'_u, \phi') = d_k(u) \tag{4.3.2}$$

Proof. The proof is by induction on k. Consider $k = 0$, i.e., before entering the loop for the first time. The only nodes of the CT which can be the starting nodes of a c-path with height zero are the accepting nodes. For every accepting node, $a \in A$, we have $d_0(a) = 0$, which is equal to the μ-updated weight of a c-path with no edges.

Assuming the inductive hypothesis, we prove that the invariants hold at the end of the $(k+1)$-th iteration of the while loop of `Lower_Bound`(M, μ). Suppose that (4.3.2) holds for every node during the k-th iteration of the while loop. If u is a switch node and during the $(k+1)$-th iteration of the while loop $d(u)$ changes in line

8, we find a node x and an edge (u, l, x), which may be added to the c-path starting at x to make a new c-path (π'_u, ϕ') such that $w_\mu(\pi'_u, \phi') = d_{k+1}(u)$. Similarly if u is a concatenation node, and during the $(k+1)$-th iteration of the while loop $d(u)$ changes in line 11, we find two edges $(u, \texttt{left}, x)$ and $(u, \texttt{right}, y)$, which may be added to the c-paths starting in x and y, respectively, to make a new c-path (π''_u, ϕ'') such that $w_\mu(\pi''_u, \phi'') = d_{k+1}(u)$, which proves (4.3.2).

In order to prove (4.3.1) by induction, we first observe that $d_{k+1}(u)$ satisfies the following properties:

- if u is a switch node, then:

$$d_{k+1}(u) \leq \min\{w_\mu(e) + d_k(x) \mid e = (u, l, x) \text{ is an edge in } M\} \tag{4.3.3}$$

- if u is a concatenation node, then:

$$d_{k+1}(u) \leq \sum\{w_\mu(e) + d_k(x) \mid e = (u, l, x) \text{ is an edge in } M\} \tag{4.3.4}$$

Indeed, both statements hold because the value of $d_{k+1}(u)$, computed by `Lower_Bound`(M, μ), is either exactly the value at the right hand side of the inequalities (4.3.3) and (4.3.4) (when all $d(x)$ values were computed in the previous iterations of the while loop), or is a smaller value (when some $d(x)$ values were decreased during the current, i.e., $(k+1)$-th, iteration).

Now, For a switch node u we use (4.3.3):

$\min\{w_\mu(\pi_u, \phi) \mid h(\pi_u, \phi) \leq k+1\} =$

$\min\{w_\mu(e) + w_\mu(\pi_x, \phi') \mid h(\pi_x, \phi') \leq k \text{ and } e = (u, l, x) \text{ is an edge in } M\} \geq$

$d_{k+1}(u)$.

Using (4.3.4) and a similar reasoning, (4.3.1) can be proved for concatenation nodes. □

Lemma 4.3.2. *If after an iteration of the while loop in* **Lower_Bound**(M, μ) *change =* **false**, *then* $d(v_{in})$ *carries the minimum* μ*-updated weight of all accepting paths of* M.

Proof. The boolean variable *change* remains `false` after an iteration of the while loop only if there is no change in $d(x)$, for any node x, during that iteration. So no additional iteration of the while loop would modify $d(x)$. By Lemma 4.3.1,

$$\forall k \geq 0, d(v_{in}) \leq \min\{w_\mu(\pi_{v_{in}}, \phi) \mid h(\pi_{v_{in}}, \phi) \leq k\}$$

and

$$\exists(\pi'_{v_{in}}, \phi') \text{ such that } w_\mu(\pi'_{v_{in}}, \phi') = d(v_{in})$$

Thus, $d(v_{in}) = \min\{w_\mu(\pi_{v_{in}}, \phi)\} = \min_{p \in P}\{w_\mu(p)\}$, where P is the set of all accepting paths of M. □

Lemma 4.3.3. *If after the last iteration of the while loop in* **Lower_Bound** *change =* **true**, *then the CT has (an infinite number of) accepting paths with negative* μ*-updated weights.*

Proof. Having *change* = `true` at the end of the last iteration of the while loop means that there is a node $v \in S \cup C$ such that $d_{|S \cup C|+1}(v) < d_{|S \cup C|}(v)$. By Lemma 4.3.1, there is a c-path $\Pi = (\pi_v, \phi)$, whose μ-updated weight is $d_{|S \cup C|+1}(v)$. Since $d_{|S \cup C|+1}(v) < d_{|S \cup C|}(v)$, the height of Π is at least $|S \cup C| + 1$. This means that there must exist two different vertices x and y with $\phi(x) = \phi(y) = u$, for some $u \in S \cup C$, such that x is an ancestor of y in π_u and if we consider the subtrees of π_u which stem from x and y as two c-paths (π'_u, ϕ') and (π''_u, ϕ''), respectively, then $w_\mu(\pi'_u, \phi') < w_\mu(\pi''_u, \phi'')$. We can replace π''_u with π'_u in the tree π_u constructing a new c-path Π' with $w_\mu(\Pi') = w_\mu(\Pi) - w_\mu(\pi''_u, \phi'') + w_\mu(\pi'_u, \phi') < w_\mu(\Pi)$. Repeating the same procedure, we can construct c-paths starting at v with negative μ-updated

weights. As v is reachable from v_{in}, we conclude that the CT has (an infinite number of) accepting paths with negative μ-updated weights. □

As a direct consequence of Lemmas 4.3.2 and 4.3.3, we can formulate the following theorem.

Theorem 4.3.1. *The procedure* `Lower_Bound`(M, μ) *returns* `true` *if and only if* μ *is a lower bound for the worst-case throughput of* M*, i.e.,* $\mu \leq wct(M)$.

In order to calculate the complexity of the `WC_Throughput_Approximation` algorithm, we notice that the algorithm executes the while loop $\log(\frac{max-min}{\epsilon})$ times. Since the while loop of the `Lower_Bound` procedure is executed $|S \cup C| + 1$ times in the worst case, and each such execution examines every edge of the CT exactly once, the total complexity amounts to $O(nm \log(\frac{max-min}{\epsilon}))$, where n and m are the number of nodes and edges in M, respectively.

4.4 Conclusions

Content inspection problems that process packets with contents from non-regular languages cannot be solved by finite state transducers. In this chapter we presented CTs to extend the range of content inspection problems which can be solved by transducers. It has been shown in [56] that CTs represent the family of simple languages which includes languages such as the simplified XML. We showed that a CT-based CIE can be efficiently implemented in hardware by adding a stack to the TLT architecture. We also investigated the problem of finding the worst-case throughput of a CT. We presented an algorithm for computing an ϵ-approximation of the worst-case throughput of a CT M. The time complexity of the algorithm is $O(nm \log(\frac{max-min}{\epsilon}))$, where n is the number of nodes, m is the number of edges, and

max (*min*) is the maximum (respectively, minimum) of the lengths of the edge labels of M.

It is worth mentioning that recently a polynomial-time algorithm for finding the exact worst-case throughput of CTs has been introduced [59]. However, from the application point of view, the linear memory requirement of our algorithm, in comparison to the cubic requirement for the algorithm of [59], makes our algorithm a preferable choice.

CTs are in fact a subclass of labeled directed And-Or graphs [60, 61]. Our algorithm as presented here actually solves a more general problem of finding an ϵ-approximation of the infimum of the mean weights of all complete hyper-paths of an And-Or graph with real weight values.

Chapter 5

Range-Matching Using TCAM

The problem of multifield packet classification can be considered as a special case of the content inspection problem where the processing of packets is limited to their headers. The policies that are used for multifield packet classification normally have a larger number of rules compared to those used for problems that involve inspection of packet contents. During the last decade there has been a continuous trend of performance advance in proposed algorithmic approaches for the multifield packet classification problem [2, 4, 62–64]. Such algorithms use off-chip random access memory in order to be scalable to databases with millions of entries.

On the other hand, most of the current and near-future's multifield packet classification databases include up to a thousand rules. As such, the industry has largely depended on TCAM for implementing multi-gigabit multifield packet classification. In the multifield packet classification application, each of the strings stored in the TCAM is a policy rule and there is an index associated to each rule showing the position of the rule inside the TCAM. The TCAM compares a given search key against all of its rules in parallel and returns the index of the rules that it matches. Having a constant search time is the main property of the TCAMs which makes them a strong medium choice for performing multifield packet classification and forwarding in the

edge-routers.

Despite the above advantages which makes TCAMs well-suited for performing high-speed packet classification, TCAMs also have limitations which reduce their efficiency. One of the limitations is that TCAMs cannot store efficiently rules that have range fields, where a range is a consecutive set of integers (e.g., $[1024, 2048]$). Traditionally a range is first mapped to a set of prefixes and then each prefix is stored in separate TCAM entry. The advantage of this method is that it is easy to implement it and also it uses a minimum-width TCAM, i.e., a TCAM as wide as the width of range-port in the policy rules. However, it does not map all ranges to TCAM rules efficiently. For example, a range of 1-65534 would be expanded into 30 TCAM rules of width 16. This problem is specially critical in the policy rules of the edge routers, where the port fields usually have ranges. In addition, as observed in [65], both the number of rules in the classification databases and the percentage of rules with one or two range fields have an increasing trend. This means that the range-expansion problem could be a serious deterrent to a complete dependent on the neat architecture of TCAMs for packet forwarding and classification in edge routers.

The interest to the problem of range matching in TCAM has recently been sharpened ([65],[66]). Despite this interest, the traditional prefix-based range expansion is still the only available method of preforming range-matching in minimum-width TCAM; Almost all available research works improve the efficiency of range matching in TCAM by some heuristics which depend on using TCAMs with widths larger than the width of the range-port in the policy rules.

In this chapter, we present a systematic approach for tackling this problem in minimum-width TCAM; We first give a formal definition for the problem of range matching in TCAM and also explain the traditional prefix-based representation of ranges in TCAM through a tree representation. We then show that replacing the

lexicographic encoding of this method with the *binary reflected Gray Encoding*, improves the expansion rate of ranges in TCAM without adding anything to the width of the TCAM.

An important problem that can provide some insight on the amount of TCAM required for range representation in minimum-width TCAM is as follows: for a fixed minimum-width encoding scheme, what is the maximum number of TCAM rules that are required for representing a range in TCAM. For a class of encoding schemes, which includes the lexicographic encoding (i.e., the standard binary encoding) and the binary reflected Gray encoding, we provide a tight lower bound on the performance of the range representation in minimum-width TCAM. Finally, we will present two linear-time algorithms that generate a minimum TCAM representation of a given range for the lexicographic encoding and the binary reflected Gray encoding.

The chapter is organized as follows. We first present a formalization of the range-representation problem in Section 5.1. We then show in Section 5.2 that using binary reflected Gray encoding instead of the lexicographic encoding improves the expansion rate of ranges in TCAM. In Section 5.3, we provide a tight lower bound on the worst-case expansion rate of ranges in minimum-width TCAM. Our two algorithms for generating a minimum TCAM representation of a given range are presented in Section 5.4. At the end of Section 5.4, we present some simulation results and the conclusions of the chapter are provided in Section 5.6.

5.1 Formalizing the Range Matching Problem

We start this section by some formal definitions for TCAMs. A TCAM can be defined as a two dimensional array of cells, where each cell carries one of the three values 0, 1, or $*$. Each row of this array is called a TCAM rule. A rule is called a *prefix* rule if all non-star symbols occur as a suffix of it. For example, both `10**` and `0110` are

prefix rules, while 0*01 is not.

A rule of a TCAM of width n is a sequence $r = e_1 e_2 \dots e_n$, where $e_i \in \{0, 1, *\}$ for $i \in \{1, \dots, n\}$, and defines the following non-empty language $L(r)$:

$$L(r) \stackrel{\text{def}}{=} L(e_1)L(e_2)\dots L(e_n),$$

where $L(0) \stackrel{\text{def}}{=} \{0\}$, $L(1) \stackrel{\text{def}}{=} \{1\}$, and $L(*) \stackrel{\text{def}}{=} \{0, 1\}$. For example, $L(0*1) = \{0\} \cdot \{0, 1\} \cdot \{1\} = \{001, 011\}$. We say that r *covers* a set of strings P if $\forall w \in P, w \in L(r)$.

A TCAM $\mathcal{R}$ consisting of k rules $r_1, r_2, \dots, r_k$ will be written as $\mathcal{R} = (r_1, r_2, \dots, r_k)$. The language of $\mathcal{R} = (r_1, r_2, \dots, r_k)$ is the union of the languages defined by its rules, i.e., $L(\mathcal{R}) \stackrel{\text{def}}{=} L(r_1) \cup L(r_2) \cup \dots \cup L(r_k)$.

Let $\mathcal{R} = (r_1, r_2, \dots, r_k)$ and $\mathcal{R}' = (p_1, p_2, \dots, p_m)$ be two TCAMs of the same width d, i.e., $r_i, p_j \in \{0, 1, *\}^d$ for $i \in \{1, 2, \dots, k\}$ and $j \in \{1, 2, \dots, m\}$, and $w \in \{0, 1, *\}^n$ be a rule of length n. We define:

$$\mathcal{R} + \mathcal{R}' \stackrel{\text{def}}{=} (r_1, r_2, \dots, r_k, p_1, p_2, \dots, p_m)\ ; \quad w\mathcal{R} \stackrel{\text{def}}{=} (wr_1, wr_2, \dots, wr_k).$$

Intuitively, the TCAM $\mathcal{R} + \mathcal{R}'$ is the union of $\mathcal{R}$ and $\mathcal{R}'$, thus its width remains d and the number of its rules is $k + m$. The TCAM $w\mathcal{R}$ is of width $d + n$ and each of its rules is the concatenation of w with a rule of $\mathcal{R}$.

Let $E : \{0, 1\}^n \hookrightarrow \{0, 1, \dots, 2^n - 1\}$ be an encoding of integer values by n-bit strings. Any subset $X \subseteq \{0, 1, \dots, 2^n - 1\}$ can be represented by a TCAM $\mathcal{R}$ of width n. More precisely, $\mathcal{R}$ represents X iff $E^{-1}(X) = L(\mathcal{R})$.

For two integers x, y, we denote by $[x, y]$ the set $\{x, x + 1, \dots, y\}$ and call it a *range*. A *prefix subset* of a range $R = [x, y]$ is defined as a subset of R including some consecutive integer values, which can be represented by a single prefix TCAM rule. For example, consider the range $[1, 9]$ over the domain $[0, 15]$. If we encode the integers of this domain by lexicographic encoding then $\{4, 5, 6, 7\}$ is a prefix subset, representable by the prefix TCAM rule 01**.

Suppose that we have a domain $D = [0, k]$, where k is a non-zero positive integer, and a list of N ranges $P = \{R_1, R_2, \ldots, R_N\}$ over D, with a priority associated to each range. The *range-matching* problem consists of finding the highest priority range of P, if any, that includes a given integer value $x, 0 \leq x \leq k$. There is a method for solving this problem using TCAM which is associated with two components: an *encoding scheme* and a *TCAM rule generation procedure.* The encoding scheme, denoted by f, is used for representing each integer value that falls within the domain D by a unique string of bits. The TCAM rule generation procedure is used to represent the ranges of P as a number of TCAM rules and load them into a TCAM; This procedure is such that, for a given value $x \in D$, the highest priority range of P that includes x, if any, can be obtained by matching $f(x)$ against the loaded TCAM.

Example 5.1.1. *Suppose that* $D = [0, 5]$ *and* $P = \{R_1, R_2, R_3\}$ *where* $R_1 = [1, 4]$, $R_2 = [3, 5]$, *and* $R_3 = [0, 2]$. *Let us first use the lexicographic representation of the integer numbers as encoding scheme. Since there are 6 integer numbers in the range D, we need 3 bits to represent all of the 6 numbers. Figure 5.1-(a) shows the lexicographic encoding for each number. Using this encoding, the three ranges can be represented by the TCAM rules as shown in Figure 5.1-(b).*

Figure 5.2-(a) shows another possible encoding scheme for the numbers in the range $[0, 5]$. *For this encoding, the TCAM rules corresponding to the three ranges are shown in Figure 5.2-(b).* ◀

Generally, there is a trade off between the width of the encoding scheme used for encoding integer values (i.e., the width of the TCAM rules), and the number of TCAM rules required for representing ranges over those values. For instance, while the second encoding scheme of Example 5.1.1 uses 2 more bits for encoding each value, the total number of TCAM rules required for representing the 3 given ranges in TCAM by this scheme is 3 rules less than that of the first encoding scheme.

value = x	encoded value = $f(x)$
0	000
1	001
2	010
3	011
4	100
5	101

(a)

range	TCAM rules
$R_1 = [1, 4]$	001 01* 100
$R_2 = [3, 5]$	011 10*
$R_3 = [0, 2]$	00* 010

(b)

Figure 5.1: (a) The first encoding scheme and (b) the TCAM rules corresponding to the three ranges of Example 5.1.1.

value = x	encoded value = $f(x)$
0	01001
1	01010
2	01100
3	10001
4	10010
5	10100

(a)

range	TCAM rules
$R_1 = [1, 4]$	01**0 100**
$R_2 = [3, 5]$	10***
$R_3 = [0, 2]$	01***

(b)

Figure 5.2: (a) The second encoding scheme and (b) the TCAM rules corresponding to the three ranges of Example 5.1.1.

For the domain $D = [0, K]$, the smallest possible width for the encoding scheme is $\lceil log_2(K+1) \rceil$, where for a positive real value c, $\lceil c \rceil$ denotes the smallest integer number that is greater than or equal to c. For this minimum width, depending on the encoding scheme, we may need a large number of TCAM rules for representing some of the input ranges. On the other hand, it is possible to represent any range over D with a single TCAM rule if we allow the width of encoding scheme to be as large as k, so that we can encode the integer values with a unary encoding. For instance, the all the 3 ranges in Example 5.1.1 can be represented by a single TCAM rule if we use the following unary encoding scheme for representing integer numbers from 0 to 5: $\{00000, 00001, 00011, 00111, 01111, 11111\}$. Using this encoding, the three ranges $[1, 4]$, $[3, 5]$, and $[0, 2]$ would be represented as `0***1`, `**111`, and `000**`, respectively.

The width of the TCAMs that are used by today's edge routers for performing multifield packet classification is limited to a few hundred bits and the ranges fields of the rules are defined over 16-bits binary values (i.e. $K = 65535$). Thus for this application, there is a constraint over the width of employed encoding scheme which makes it indispensable to look for efficient encoding schemes under some width constraint. In order to formalize the problem of range matching in TCAM under a width constraint on the length of TCAM rules, we define the *maximum expansion rate* of an encoding scheme f over the domain $D = [0, K]$, as the maximum number of TCAM rules which are required for representing any range over D in TCAM and denote it by $E(f)$. Note that there must be a range over D which cannot be represented by $E(f) - 1$ or less TCAM rules.

Definition 5.1.1. The problem of *range matching in width-limited TCAM* consists in finding an encoding scheme for the integer values of D, denoted by f_{opt}, whose width is subjected to a fixed upper-bound and has a minimum $E(f_{opt})$. In order to be able to use such encoding scheme for representing and loading ranges into TCAM rules, f_{opt} should be also accompanied with a TCAM rule generation procedure.

Finding an optimal solution for the problem of range matching in TCAM , i.e., finding an optimal encoding scheme and its associated TCAM rule generation procedure, appears to be very challenging. However, any non-optimal solutions for this problem which yields good maximum (and/or average) TCAM expansion rates compared to the traditional methods, would have an immediate application in practice. In next section we first look at the only available method of representing ranges in minimum-width TCAM. We then show that we can get better expansion rates by replacing the encoding scheme of this method with the reflected binary Gray encoding.

5.2 Range Matching by Minimum-Width Encoding Schemes

The only available solution for the problem of range matching in minimum-width TCAM is called prefix-based expansion of ranges. We proceed in this section by first explaining the TCAM rule generation procedure of this method which uses the lexicographic encoding. We then explain how we can improve the expansion rates of range matching by changing the encoding scheme of the prefix-based expansion method.

5.2.1 Prefix-Based Expansion of Ranges

Ranges are traditionally represented in TCAMs by splitting each range into a number of prefix subsets and then representing each prefix subset with a single TCAM rule. This method, which uses the lexicographic encoding of numbers as its encoding scheme, can be easily explained by the binary tree representation of lexicographic encoding. A binary tree representation of n-bit binary numbers has 2^n leaves and each of the 2^n paths from the root of the tree to a leaf corresponds to one of the binary

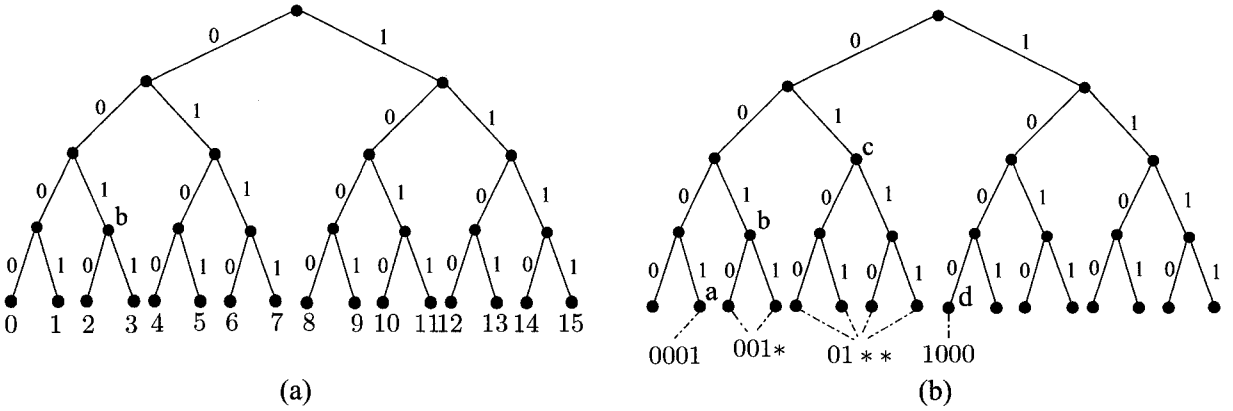


Figure 5.3: (a)- The binary tree corresponding to 4-bit lexicographic encoding. (b)- Representing the range [1, 8] in TCAM by the prefix-based expansion method.

numbers. The correspondence between a path and a binary number is reflected on the binary tree by the edge-labels of the path. Figure 5.3-(a) shows the binary tree corresponding to 4-bit lexicographic encoding. For example all the edges on the path from the root of this tree to its rightmost leaf are labeled with 1 which means this leaf corresponds to binary number 1111. Notice that we can represent some sets of successive binary values by a single TCAM rule. For instance, the set $B = \{0010, 0011\}$ can be represented by concatenating the edge labels of the path from root to node b and putting $*$ at the end of the rule to get `001*`. This observation is the stepping stone for the prefix-based expansion of ranges.

The prefix-based expansion of ranges is formally defined as finding a minimum number of mutually exclusive prefix subsets that cover a range R and then mapping each prefix subset to its corresponding prefix TCAM rule.

Example 5.2.1. *Suppose that we want to represent the range* $[1, 8]$ *in TCAM. Figure 5.3-(b) represents the prefix subsets of this range, which each can be represented by a single prefix rule, and their corresponding node on the tree. Using this graph, this range is expanded to the following set of prefix rules:* {`0001,001*,01**,1000`}. ◀

The maximum expansion rate for the prefix-based expansion of an n-bit range is $2n-2$ TCAM rules. This expansion rate is the result of representing the range $[1, 2^n - 2]$ by prefix rules. For instance, in Figure 5.3-(a), the range $[1, 14]$ can be represented by the following 6 TCAM rules: {0001,001*,01**,10**,110*,1110}. However, if we notice that the range $[1, 2^n - 2]$ includes all the $n-$bit binary numbers except for the all 0's and the all 1's n-bit numbers, then we can represent this range by n TCAM rules. Using this technique, the range $[1, 14]$ can be represented by 4 rules as follows: {01**, *01*, **01, 1**0}. Hence, by a small change in the TCAM rule generation procedure of the prefix-based expansion method to treat the range $[1, 2^n - 2]$ differently, we can improve the maximum expansion rate for an n-bit range, $n \geq 3$, to be at most $2n - 3$. This observation emphasizes the fact that both the encoding scheme and the TCAM rule generation procedure can affect the performance of a method of range representation in TCAM.

In the following subsection we show that replacing the encoding scheme of the prefix-based method with the binary reflected Gray encoding provides us with a gain in the average number of rules that are stored in TCAM for each range, without increasing the width of the TCAM rules.

5.2.2 Range Expansion Using Binary Reflected Gray Encoding

In the prefix-based range expansion method, each number is represented by its lexicographic encoding. It is likely that using a different binary encoding of the numbers yields a smaller maximum (and/or average) expansion rate for the TCAM representation of ranges. In order to show this potential, we consider one of the most well-known families of binary encoding schemes: *Gray Codes.*

Gray codes are a family of encoding schemes where each Gray code lists all the

numbers in the range of $[0, 2^n - 1]$, with n-bit numbers, such that successive numbers (including the first and the last) differ in exactly one position. The most famous example of Gray codes is the *binary reflected Gray code* ([67], [68]) which can be explained as follows. Suppose that L_n denotes the listing for n-bit numbers and L_1 is the list 0, 1. Then for $n > 1$, L_n is formed by pre-pending a bit of '0' to every number in the list L_{n-1}, and following that list by the reveres of L_{n-1} with a bit of '1' prepended to every number. So, for example, $L_2 = 00, 01, 11, 10$ and $L_3 = 000, 001, 011, 010, 110, 111, 101, 100$.

Binary reflected Gray encoding of numbers can be represented by a binary tree. Figure 5.4-(a) shows the binary tree for L_4. Note that in comparison to Figure 5.3-(a), some of the edge labels have been flipped on this graph. For example the edges on the path from the root of this tree to its rightmost leaf are labeled with with 1000, i.e., the 4-bit binary reflected Gray encoding for 15.

A property of this encoding, which helps us to devise an efficient TCAM rule generation procedure for this encoding, can be explained through the vertical dashed lines that have been depicted in this figure; Let us begin with the vertical dashed-line which goes through the root of the tree. If two equal-size prefix subsets of the tree are in a symmetric position with respect to this line, then we can represent both of these two prefix subsets with a single TCAM rule. For example the two prefix subsets $\{1\}$ (i.e., the leaf with encoding 0001) and $\{14\}$ (i.e., the leaf with encoding 1001) are symmetric with respect to this dashed-line and can be explained by the single rule $*001$. The other dashed lines of the figure have a similar property. For instance the two subsets $\{2, 3\}$ and $\{4, 5\}$ are symmetric with respect to the second dashed line from left and can be expressed by the single rule $\{0*1*\}$. Note that in the prefix-based expansion method we represent each of these subsets with a different TCAM rule. The two ranges considered in the following example show how this property of the binary reflected Gray encoding helps us to represent a range with a smaller

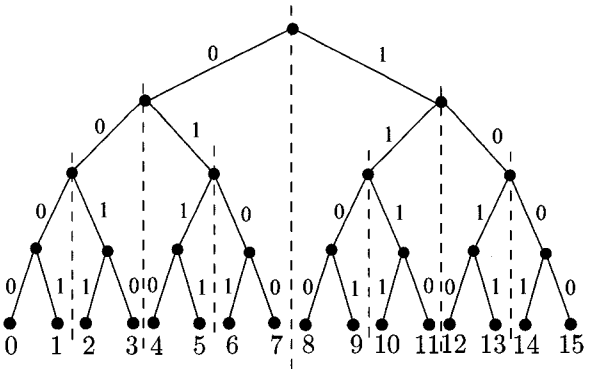


Figure 5.4: The binary tree corresponding to binary reflected Gray encoding of the numbers between 0 and 15.

number of TCAM rules.

Example 5.2.2. *Let us first compare the number of TCAM rules required for representing the range* $[1, 13]$ *in binary reflected Gray encoding, with that of the prefix-based method. As it is illustrated in Figure 5.5-(a), TCAM representation of the range* $[1, 13]$ *by binary reflected Gray encoding needs these 3 rules:* $\{0001, *01*, *1**\}$. *The prefix-based method, however, represents the same range with the following 5 rules:* $\{0001, 001*, 01**, 10**, 110*\}$.

Figure 5.5-(b) shows how we can use the symmetric property of binary reflected Gray encoding to represent the range $[1, 8]$ *by 3 TCAM rules as* $\{0*01, 0*1*, *100\}$. *Using the prefix-based method, this range is represented by the following 4 TCAM rules:* $\{0001, 001*, 01**, 1100\}$. ◀

It can be easily verified that for every range, the number of TCAM rules obtained from the range representation method that uses binary reflected Gray encoding is smaller than or equal to the number of TCAM rules obtained from the prefix-based method.

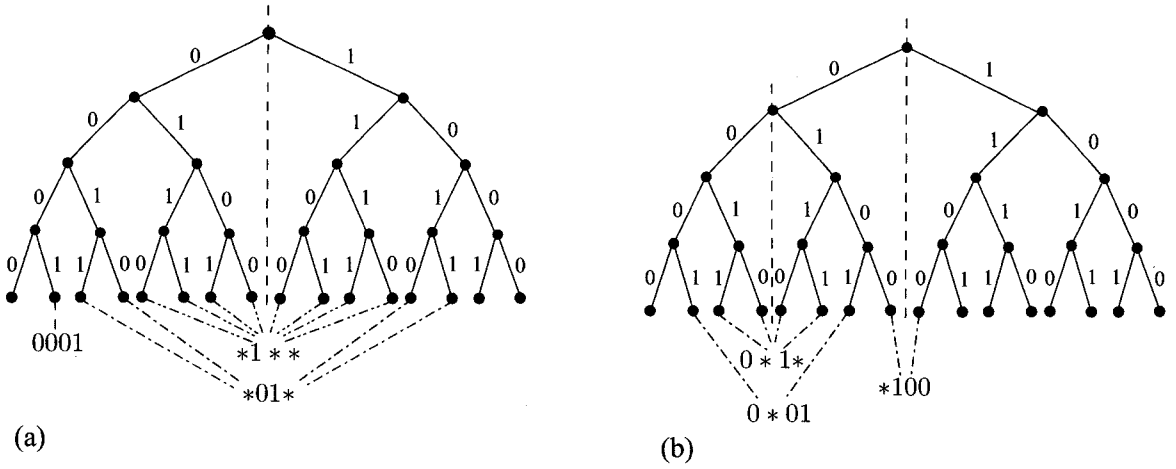


Figure 5.5: Generating TCAM rules, using the symmetry property of binary reflected Gray encoding, for the ranges (a) $[1, 13]$ and (b) $[1, 8]$.

5.3 Maximum Expansion Rate of Minimum-Width Encoding Schemes

One would expect that the worst-case expansion rate of range representation in TCAM depends on both the encoding scheme and the TCAM rule generation procedure. In this section we show that for a class of minimum-width encoding schemes, which includes the lexicographic encoding and the binary reflected Gray encoding, the maximum expansion rate cannot be smaller than a tight bound.

5.3.1 Definitions and Notations

We start by formal definitions of the concepts that we will use for proving the existence of the above-mentioned tight bound and also throughput the next section. By $T_\emptyset$ and $\bullet$ we will represent the empty tree and the single node tree, respectively. Let T be a tree and $w \in \{0,1\}^*$. By $w^{-1}T$ we will denote the corresponding sub-tree of T; the root of $w^{-1}T$ is the last vertex of the unique path starting from the root of T and

labeled by w. For a non-empty tree T, by $h(T)$ we denote the height of T, i.e., the length of the longest path from the root to a leaf. For $a \in \{0,1\}$, by $\overline{a}$ we will denote the *complement* of a, i.e., $\overline{0} \stackrel{\text{def}}{=} 1$ and $\overline{1} \stackrel{\text{def}}{=} 0$.

We define a *full* tree of height n as a perfect binary tree of height n such that each pair of sibling edges are labeled 0 and 1. The assignment of labels to the edges (two alternatives per each internal node) can be chosen arbitrarily. Let T_n be a full tree of height n. The label $w \in \{0,1\}^n$ of the path from the root to i-th leaf defines the n-bit encoding of number i, with 0 corresponding to the furthest left leaf of the tree. In that way, T_n defines a bijection $T_n : \{0,1\}^n \hookrightarrow \{0,1,\ldots,2^n-1\}$, called an n-bit *dense-tree encoding*. The lexicographic encoding and the binary reflected Gray encoding are two important examples of dense-tree encodings.

In the context of a dense-tree encoding T_n, a set $X \subseteq \{0,1\}^n$ defines both the set $T_n(X)$ of integers and a subset of leaves of T_n. The set X can be represented by a *skeleton tree* (see Figure 5.6) that is obtained from T_n by:

(1) Obtaining a range tree by removing all edges which are not leading to the leaves of X;

(2) Turning into leaves all full subtrees of the range tree.

We say that a tree T is a *chain*, if every vertex of T has at most one non-leaf child. A *double-chain* is a tree with at most one vertex v having two non-leaf children and such that all ancestors of v have only one child. Examples of a chain and a double-chain are illustrated in Figure 5.7.

A set of binary strings X of length n is a *range-set* of a dense-tree encoding T_n if $T_n(X)$ is a range. A chain is called *left-chain* (resp., *right-chain*) if it represents a range-set and every leaf node of it is a right (resp., left) child of its parent. Examples of a left-chain and a right-chain are shown in Figure 5.8. Intuitively, a left-chain C_L defines an range $[x, 2^k-1]$, and a right-chain C_R a range $[0,y]$ or $[2^k,y]$, where $k \leq n$,

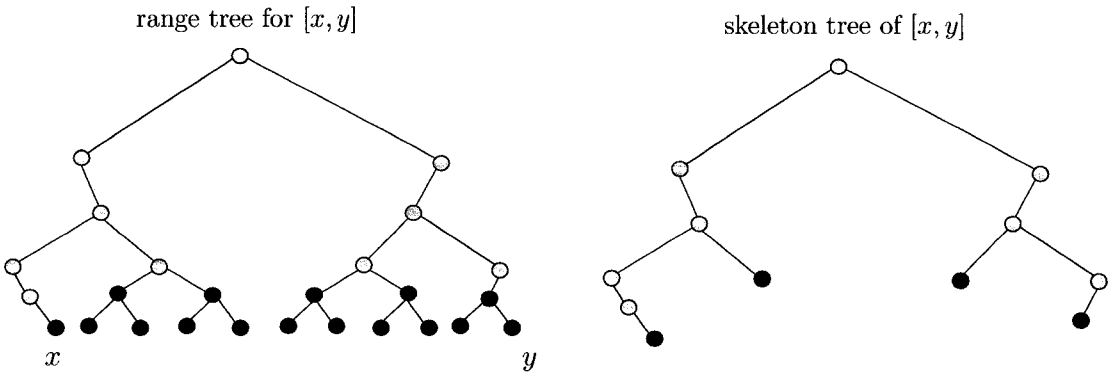


Figure 5.6: The range tree and skeleton tree of a range $[x, y]$.

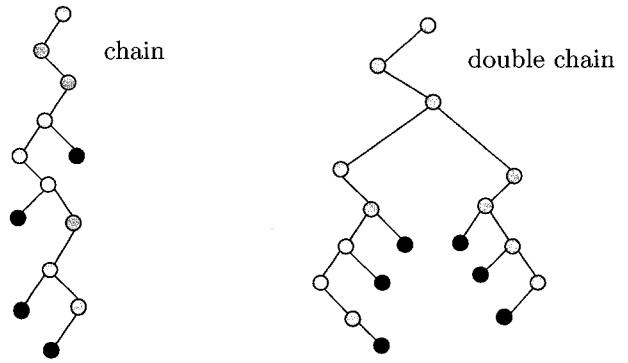


Figure 5.7: A chain and a double-chain with 4 and 6 leaves, respectively.

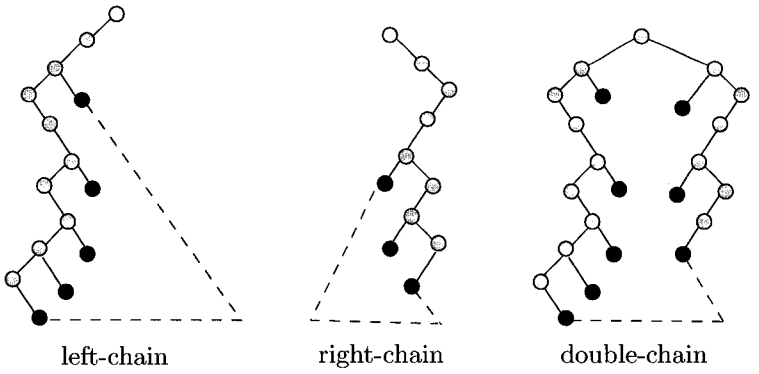


Figure 5.8: Three types of skeleton trees: left-chain, right-chain, and double-chain.

n is the width of the dense-tree encoding, and $x, y \in \{0, 1, \ldots, 2^n - 1\}$. We say that C_L and C_R are *complementary chains*, if and only if $x \leq y + 1$, i.e., the union of their corresponding ranges yields $[0, 2^n - 1]$, assuming $n \geq \max\{h(C_L), h(C_R)\}$.

5.3.2 Preliminary Results

We first show through the following two lemmas that on a dense-tree encoding, the skeleton tree corresponding to any range is a double-chain.

Lemma 5.3.1. *Let $X \subseteq \{0,1\}^n$. The following statements are equivalent:*

1. *There exists a dense-tree encoding T_n for which $T_n(X) = [0, |X| - 1]$;*
2. *There exists a dense-tree encoding T_n for which $T_n(X) = [2^n - |X|, 2^n - 1]$;*
3. *For any dense-tree encoding the skeleton tree of X is a chain.*

Proof. The proofs of all equivalences except for $1 \Rightarrow 3$ and $2 \Rightarrow 3$ are obvious. We present a proof based on induction on n for $1 \Rightarrow 3$; The proof for $2 \Rightarrow 3$ is very similar. For $n = 1$ any subset of $\{0, 1\}$ is a range, and any skeleton tree of height not bigger than 1 is a chain.

Suppose that the induction hypothesis holds for all $k \leq n$. For $X \subseteq \{0,1\}^{n+1}$, we can write $X = X_0 \cup X_1$, where X_0 and X_1 are the sets of all those strings of X which start by 0 and 1, respectively. There are two cases:

1. If X_0 or X_1 is empty, then the initial symbol is the same for all elements of X and it can be omitted. Thus, the proof is complete as the induction hypothesis applies directly.

2. Suppose that none of X_0 or X_1 is empty and consider an arbitrary dense-tree encoding T_{n+1}. Since there exists a dense-tree encoding T'_{n+1} for which $T'_{n+1}(X) = [0, |X| - 1]$, we must have $|X| > 2^{n-1}$ and also either X_0 or X_1 must include 2^{n-1} strings. If we assume without loss of generality that $|X_0| = 2^{n-1}$, then the skeleton tree of X_0 in T_{n+1} is just an edge labelled by 0. Besides, by the hypothesis induction, the skeleton tree of X_1 is a single edge connected to a chain. Thus, the root of the skeleton tree of X in T_{n+1} has a leaf as one child and a chain connected to the other child which means that the skeleton tree is a chain itself.

□

Lemma 5.3.2. *Let $X \subseteq \{0,1\}^n$. The following statements are equivalent:*

1. *There exists a dense-tree encoding T_n for which $T_n(X)$ is a range;*

2. *For any dense-tree encoding the skeleton tree of X is a double-chain.*

Proof. The proof of $2 \Rightarrow 1$ is obvious. We prove the $1 \Rightarrow 2$ part by induction on n. For $n = 1$ see the proof of Lemma 5.3.1.

Suppose that the induction hypothesis holds for all $k \leq n$. For $X \subseteq \{0,1\}^{n+1}$, we can write $X = X_0 \cup X_1$, where X_0 and X_1 are the sets of all those strings of X which start by 0 and 1, respectively. There are two cases:

1. If X_0 or X_1 is empty, then the initial symbol is the same for all elements of X and it can be omitted. Thus, the proof is complete as the induction hypothesis applies directly.

2. Suppose that none of X_0 or X_1 is empty and consider an arbitrary dense-tree encoding T_{n+1}. Since there exists a dense-tree encoding T'_{n+1} for which $T'_{n+1}(X)$ is a range, by Lemma 5.3.1, the skeleton trees of X_0 and X_1 in T_{n+1} are chains. Thus, only the root of the skeleton tree of X in T_{n+1} has two non-leaf children which means it is a double chain.

□

The complexity of building the skeleton tree of a given range on the n-bit lexicographic encoding is linear in n. Therefore, we can have the following corollary from the result of Lemma 5.3.2:

Corollary 5.3.1. *Let $X \subseteq \{0,1\}^n$. There is a linear-time algorithm to test whether there exists a dense-tree encoding T_n for which $T_n(X)$ is a range.*

In continuation of our preliminary results, we determine the minimum size of a TCAM that represents a given range whose skeleton tree is a chain. The proof of the following lemma is immediately implied from the definition of a prefix rule and prefix subsets:

Lemma 5.3.3. *Let $X \subseteq \{0,1\}^n$ be a range-set of a dense-tree encoding T_n. The minimum number of prefix rules covering X equals the number of leaves in the skeleton tree of X in T_n.*

Theorem 5.3.1. *Let the skeleton tree of $X \subseteq \{0,1\}^n$ in a dense-tree encoding T_n be a chain with k leaves. The minimal size of a TCAM representing X is k.*

Proof. Using Lemma 5.3.3 it is clear that k rules are sufficient for representing X, each rule representing a leaf of the chain.

We have to prove that we need at least k rules (prefix or non-prefix) to represent X. Assume, by contradiction, we can represent X by k' rules $e_1, e_2, \ldots, e_{k'}$, with $k' < k$. Let $d = d[1]d[2]\ldots d[n]$ be the label of the path from the root to the sibling of the bottom leaf of the chain, which means that $d \notin X$. Each rule e_i has to have at least one position p_i such that $e_i[p_i]$ is $\overline{d[p_i]}$; we choose always the first such position. Since $k' < k$, there is a position r which corresponds to an incoming edge of a leaf of the chain and which was not selected by any of positions p_i. We change the symbol on this position of d to $\overline{d[r]}$. Then the resulting string belongs to the range but it is not covered by any of the rules. □

Corollary 5.3.2. *In an n-bit dense-tree encoding, representing each of the ranges $[1, 2^n - 1]$ and $[0, 2^n - 2]$ needs exactly n TCAM rules.*

Consider a set $X \subseteq \{0,1\}^n$. A TCAM rule r is called an *X-limited* rule if it does not cover any string out of X, i.e., $L(r) \subseteq X$. An X-limited rule r is said to be *X-essential* if there is no other X-limited TCAM rule that covers r. Any coverage of X by a TCAM with k rules can be turned into a coverage by k X-essential TCAM rules. Therefore, it is quite natural to consider only X-essential TCAM rules in the process of finding a minimal TCAM representation of X; representation of a set in TCAM by essential rules will be called a "canonical representation". Note that for a given set X, there may be many minimal canonical representations of X.

In the context of two-level logic generation, an X-essential TCAM rule is called a "prime implicant" of X (see [69]). Prime implicants play an important role in the process of two-level logic generation. In general, the number of prime implicants for a given set may be exponential with respect to the number n of input bits [69]. In the following, we prove that in the cases of the n-bit lexicographic encoding and n-bit

reflected Gray encoding, a range-set $X \subseteq \{0,1\}^n$ admits no more than $n(n-1)+1$ and $2n$ X-essential TCAM rules, respectively.

Lemma 5.3.4. *Let $X \subseteq \{0,1\}^n$ such that the skeleton tree of X in a dense-tree encoding T_n is a chain with $k \leq n$ leaves. There are exactly k different X-essential TCAM rules.*

Proof. It can be verified that every X-essential rule is of form $s_1 s_2 \dots s_i \dots s_n$, where i is the level of a leaf in the chain and, for $1 \leq j \leq n$,

$$s_j = \begin{cases} \overline{d[i]} & \text{if } i = j \\ d[j] & \text{if } j < i \text{ and there is no leaf at level } j \\ * & \text{otherwise} \end{cases}$$

where $d = d[1]d[2]\dots d[n]$ is the path from the root to the sibling of the bottom leaf of the chain. □

Theorem 5.3.2. *Let $X \subseteq \{0,1\}^n$ be a range-set in the n-bit lexicographic encoding $\mathtt{Lex}_n$. There is at most $n(n-1)+1$ different X-essential TCAM rules.*

Proof. Let $\mathtt{Lex}_n(X) = [x, y]$. The proof is by induction on n. Suppose that $\mathtt{Lex}_n(0u) = x$ and $\mathtt{Lex}_n(1v) = y$ for some $u, v \in \{0,1\}^{n-1}$. The easy case when the encodings of x and y are starting by the same digit is skipped.

The set $P([\mathtt{Lex}_n(0u), \mathtt{Lex}_n(1v)])$ of all X-essential TCAM rules can be split into three disjoint sets P_0, P_1, and P_*, where P_a, $a \in \{0, 1, *\}$, denotes the set of all X-essential TCAM rules that start with a. By Lemma 5.3.4, we have: $|P_0| \leq n-1$ and $|P_1| \leq n-1$. Also $|P_*| = |P([\mathtt{Lex}_n(u), \mathtt{Lex}_n(v)])|$, where the range $[\mathtt{Lex}_n(u), \mathtt{Lex}_n(v)]$ is encoded by $n-1$ bits. Thus, $p(n) \leq 2(n-1) + p(n-1)$, with $p(1) = 1$, where $p(i)$ denotes the maximum number of different X-essential TCAM rules for any range I with $\mathtt{Lex}_i(X) = I$. □

Theorem 5.3.3. *Let $X \subseteq \{0,1\}^n$ be a range-set in the reflected Gray encoding $\texttt{Gray}_n$. There is no more than $2n$ different X-essential TCAM rules.*

Proof. Let $\texttt{Gray}_n(X) = [x, y]$ be such that encodings of x and y differ in the first digit. The easy case when the encodings of x and y begin by the same digit is skipped. In reflected Gray encoding, the dense-tree is "reflective", i.e., the labels of the right-hand side and the left-hand side subtrees rooted at every node, are the mirror copies of each other. Suppose that $x < y$, x is encoded as av, y as $\overline{a}u$, $a \in \{0,1\}$, and $u, v \in \{0,1\}^{n-1}$. Then by reflective property, au encodes $2^n - 1 - y$.

In case when $x \leq 2^n - 1 - y$, there is no X-essential TCAM rule which starts by $\overline{a}$, and symmetrically, if $x \geq 2^n - y$ then there is no X-essential TCAM rule which starts by a. Without loss of generality we may assume that $x \leq 2^n - 1 - y$. Therefore, X-essential TCAM rules can be split into two sets P_a (i.e., the set of rules starting by a), and P_* (i.e., those rules that start by $*$). Every rule from P_a with initial a removed must be an essential TCAM rule for range $[x, 2^{n-1} - 1]$ on $n-1$ bits, and every rule from P_* with initial $*$ removed must be an essential TCAM rule for range $[2^n - 1 - y, 2^{n-1} - 1]$ on $n-1$ bits. Since the skeleton trees of both these ranges are chains (or empty), by Lemma 5.3.4 each of them has at most $n-1$ (or none if empty) different essential TCAM rules. □

5.3.3 Maximum Expansion Rate for Dense Tree Encoding Schemes

We first show that, for all n-bit dense-tree encoding schemes, any range over $D = [0, 2^n - 1], n \geq 1$, can be represented by $\max(n, 2n-3)$ or less TCAM rules. We then show that in cases of the lexicographic and the binary reflected Gray encodings, the maximum expansion rate is actually $2n - 4$.

As it was proved in Lemma 5.3.3, the minimum number of prefix rules required for

representing a range in TCAM is equal to the number of leaves in its corresponding skeleton tree. For some ranges, this number can be significantly reduced by using non-prefix TCAM rules.

Lemma 5.3.5. *There is a dense-tree encoding T_n and a range I such that the minimum number of prefix rules representing I is $2n-2$ and the minimum TCAM size is only n.*

Proof. Consider the range $I = [1, 2^n - 2]$ in the lexicographic encoding on n bits. The skeleton tree for $X \subset \{0,1\}^n$ such that $\mathtt{Lex}_n(X) = I$ has $2n-2$ leaves and thus by Lemma 5.3.3 it cannot be covered by less than $2n-2$ prefix rules. However X can be covered by the following n rules: $X = \{rot^k(01*\ldots*) \mid 0 \leq k < n\}$, where rot denotes the left-rotation of a word, i.e., $rot(aw) \stackrel{\mathtt{def}}{=} wa$, for $a \in \{0,1,*\}$ and $w \in \{0,1,*\}^{n-1}$. □

Theorem 5.3.4.

(a) *Let T_n be a lexicographic or reflected Gray encoding of length n. There is a range $I = [x, y]$ which cannot be represented with less than $\max(n, 2n-4)$ TCAM rules.*

(b) *For each $n \geq 1$ there exists a dense-tree encoding T_n and a range $I = [x, y]$ which needs $\max(n, 2n-3)$ TCAM rules.*

Proof.

Point (a). For $1 \leq n \leq 3$, we have $\max(n, 2n-4) = n$ and the theorem's claim can be easily proved using Corollary 5.3.2. For $n \geq 4$, we provide a proof for the lexicographic encoding; the proof for the reflected Gray encoding is similar to this proof. Consider the range $I = [r_l, r_h] = [\mathtt{Lex}_n(s_l), \mathtt{Lex}_n(s_h)] = [\mathtt{Lex}_n(0100\ldots001), \mathtt{Lex}_n(1011\ldots110)]$ whose corresponding range-set has a skeleton tree as illustrated in Figure 5.9(A) for the lexicographic encoding. The encoding of r_l, i.e., s_l, starts with 01, followed by $n-3$ zeros and ends by 1. The encoding of r_h, i.e., s_h, starts

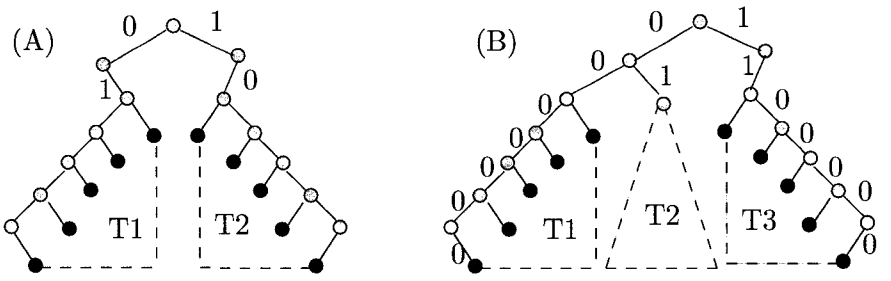


Figure 5.9: The lexicographic tree in (A) needs at least $2n-4$ rules for lexicographic coding, the (non-lexicographic) tree in (B) needs at least $2n-3$ TCAM rules.

with 10, followed by $n-3$ ones and ends by 0. We divide this range to two sub-ranges $I_1 = [r_l, 2^{n-1}-1] = [\mathtt{Lex}_n(0100\ldots001), \mathtt{Lex}_n(0111\ldots11)]$ and $I_2 = [2^{n-1}, r_h] = [\mathtt{Lex}_n(100\ldots00), \mathtt{Lex}_n(1011\ldots110)]$. The encodings of all values in the range I_1 start with 01 and thus they are within the first half of the domain $[0, 2^n-1]$. This means that all the leaves corresponding to the values in I_1 fall on the left side of the vertical line that passes through the root of the tree. The encodings for the values of range I_2, however, start with 10 and so their corresponding leaves fall on the right side of the vertical line. This means that any TCAM rule that represents a value from I_1 cannot represent any value from I_2 and vice verse. As such, the minimum number of TCAM rules required for representing I is equal to the addition of the minimum number of rules that are required to represent each of the ranges I_1 and I_2, separately.

Since the encodings of all the values from I_1 start with 10, we can represent this range by prepending 10 to the $n-2$ rules that are required to represent the range $[1, 2^{n-2}-1] = [\mathtt{Lex}_n(00\ldots001), \mathtt{Lex}_n(11\ldots11)]$ over the domain $[0, 2^{n-2}-1]$. Hence, using Corollary 5.3.2, we need at least $n-2$ TCAM rules to represent this range. Similarly, it can be seen that representing I_2 needs at least $n-2$ TCAM rules. This means that the range I cannot be represented by less than $2n-4$ TCAM rules.

Point (b). An example of such a dense-tree encoding T_n and the range $[1, 2^{n-1}+$

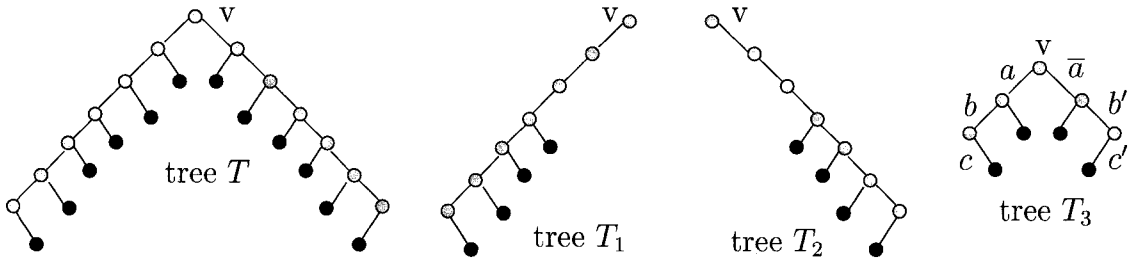


Figure 5.10: The case when skeleton tree has maximal number of prefix rules $2n-2$. T is decomposed into T_1, T_2, and T_3. For T_1 and T_2 we use $2n-6$ prefix rules. For T_3 three general rules are enough (instead of 4 prefix rules).

$2^{n-2}-2]$ is shown in Figure 5.9(B). □

Theorem 5.3.5. *Let T_n be a dense-tree encoding of length n. Every range $I=[x,y]$ can be represented by* $\max(n, 2n-3)$ *TCAM rules.*

Proof. If $1 \leq n \leq 3$, we have $\max(n, 2n-3) = n$ and it can be checked manually that every range can be represented by n or less TCAM rules. For $n > 3$, if the skeleton tree of I has $2n-3$ or less leaves, then by Lemma 5.3.3 I can be represented by $2n-3$ or less prefix TCAM rules. The skeleton tree has maximal number of $2n-2$ leaves iff it has a shape similar to tree T of Figure 5.10. We decompose this tree into three subtrees T_1, T_2, and T_3. The subtrees T_1 and T_2 are chains with $n-3$ leaves each, and thus altogether they need $2n-6$ TCAM rules. It is enough to show that T_3 needs only 3 TCAM rules. Let abc ($\overline{a}b'c'$) be the labels of the left (resp., right) branch of T_3, as shown in Figure 5.10. We consider all possible cases as follows. If $b'c' = \overline{b}\overline{c}$, then rules $*\mathtt{b}\overline{\mathtt{c}}$, $\overline{\mathtt{a}}*\mathtt{c}$, and $\mathtt{a}\overline{\mathtt{b}}*$ represent T_3; If $b'c' = \overline{b}c$, then rules $**\overline{\mathtt{c}}$, $\overline{\mathtt{a}}\mathtt{bc}$, and $\mathtt{a}\overline{\mathtt{b}}\mathtt{c}$ represent T_3; If $b'c' = bc$, then rules $*\overline{\mathtt{b}}\overline{\mathtt{c}}$, $\mathtt{a}\overline{\mathtt{b}}\mathtt{c}$, and $\mathtt{ab}\overline{\mathtt{c}}$ represent T_3. □

Theorem 5.3.6. *Let $T_n \in \{\mathtt{Lex}_n, \mathtt{Gray}_n\}$ be the lexicographic or the reflected Gray encoding of length n. Every range $I=[x,y]$ can be represented by* $\max(n, 2n-4)$ *TCAM rules.*

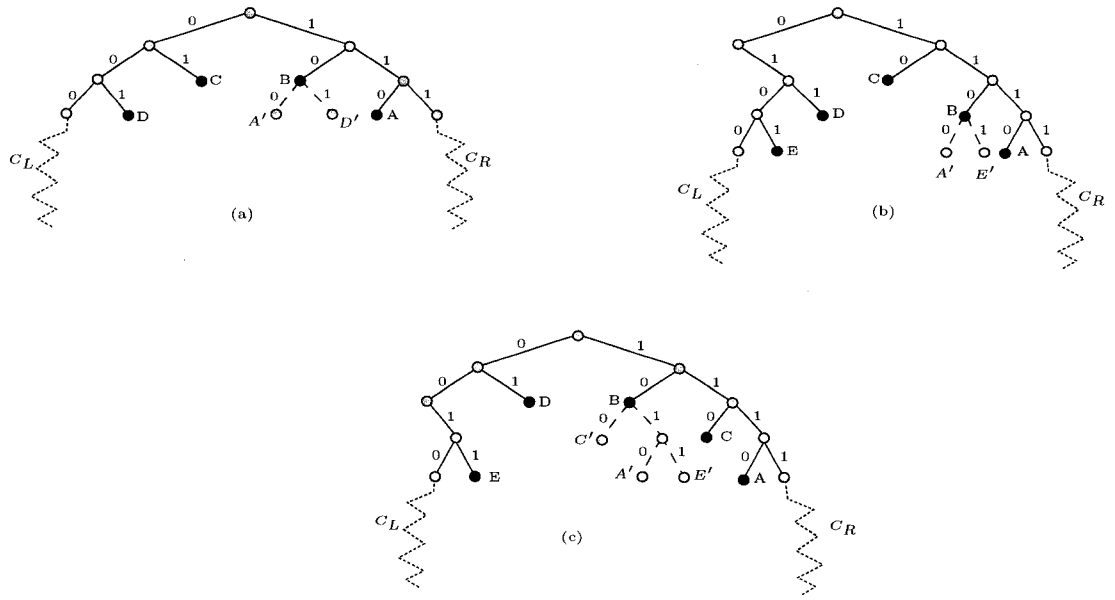


Figure 5.11: Three possible forms of a skeleton tree with $2n-3$ leaves, for $n \geq 5$.

Proof. We only show the proof for the lexicographic encoding; the proof for the Gray encoding is similar. For $n \leq 4$, it can be manually verified that every range I can be represented by n or less TCAM rules. For $n \geq 5$, if the range tree has $2n-2$ leaves, then by proof of Lemma 5.3.5 it can be represented by n TCAM rules. If the skeleton tree has $2n-3$ leaves, then it has one of the formats shown in Figure 5.11, where C_L and C_R are left-chain and right-chain, respectively. In all these three cases it can be proved that the leaf labeled by letter B can be represented by a combination of the TCAM rules that represent the other labeled leaves and thus (by Lemma 5.3.3) the whole range can be represented by $2n-4$ TCAM rules. For instance, in the skeleton tree of Figure 5.11-(a), changing the second bit and the first bit of the prefix rules that represent the leaves A and D, respectively, to $*$ results in two rules that represent A' and D', either. As such, representing leaf B does not need a separate TCAM rule. □

5.4 Ranges in Gray and Lex encodings

In this section we present two very fast algorithms for generating minimal canonical TCAM representations of ranges. The first algorithm deals with the reflected Gray encoding and the second one with the lexicographic encoding. If the skeleton tree of a given range is a chain C, we can directly use the algorithm implied by Lemma 5.3.4 to calculate its unique minimal canonical TCAM representation, $\mathcal{C}_n(C)$, independently of the dense-tree encoding T_n. If the skeleton tree is not a chain, we use two different approaches to obtain minimal canonical TCAM representations for the reflected Gray encoding and the lexicographic encoding.

5.4.1 Ranges in the reflected Gray encoding

Unlike for the lexicographic encoding, not every subtree of the reflected Gray dense-tree encoding $\texttt{Gray}_n$ is a reflected Gray dense-tree encoding. However, every pair of sibling sub-trees of the dense tree representing $\texttt{Gray}_n$ are the mirror copies of each other, i.e., $\texttt{Gray}_k$ and its mirror copy $\overline{\texttt{Gray}}_k$, $1 \leq k < n$.

Suppose that $I = [x, y]$ is a range and $y < 2^n$. We say that I is *reciprocal* if $x = 2^n - 1 - y$. If I is reciprocal, then there is a $w \in \{0,1\}^{n-1}$ such that $x = \texttt{Gray}_n(0w)$ and $y = \texttt{Gray}_n(1w)$. (The same holds for $\overline{\texttt{Gray}}_n$ where 0 and 1 are interchanged.)

Lemma 5.4.1. *Let $I = [x, y]$ be a reciprocal range with $x = \texttt{Gray}_n(0w)$ and $y = \texttt{Gray}_n(1w)$. Let $\mathcal{R}$ be a minimal canonical TCAM representation of the range $I' = [\texttt{Gray}_{n-1}(w), 2^{n-1} - 1]$. The TCAM $*\mathcal{R}$ is a minimal canonical TCAM representation of the range I. (See Figure 5.12)*

Our algorithm for calculating a minimal canonical TCAM representation of ranges in Gray encoding relies on Lemma 5.4.1; it divides a given non-reciprocal range into two overlapping sub-ranges where one of the ranges is reciprocal and the other one is

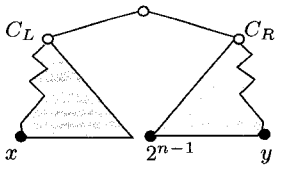


Figure 5.12: Illustration of Lemma 5.4.1; y is the mirror image of x with respect to the root of the tree. If $\mathcal{R}$ is a minimal TCAM for left-chain C_L then $*\mathcal{R}$ is a minimal TCAM for the whole tree.

not. Then it generates the minimal TCAM representation of the range as the union of those of the sub-ranges.

We first present the algorithm in the form of a pseudo-code and then provide a detailed description of it. The procedure GrayTCAM($\mathcal{S}, n$) of Figure 5.13 generates a minimal canonical TCAM representation of a range $[x, y]$ whose skeleton tree $\mathcal{S}$ has the form of Figure 5.14. In this skeleton tree y' is the mirror image of y in the left subtree and $y' > x$. Depending on the values of d_1 and d_2, the computation of the minimal TCAM is reduced to the computation of a minimal TCAM for the right-chain C_1 and one of the left-chains C_2 or C_3, where C_1, C_2, and C_3 are illustrated in Figure 5.15. In the GrayTCAM($\mathcal{S}, n$) procedure, $LCA(x, y')$ denotes the lowest common ancestor of x and y', which can be easily calculated by taking the greatest common prefix of x and y' in $\mathtt{Gray}_n$. $\Pi_L(u, v)$ is the label of the path from the left child of the root of $\mathcal{S}$ to node v, where every symbol of a left-going edge is replaced by $*$.

We now present a more detailed description of the algorithm. Let us denote by $\vec{z} = z_1 z_2 \ldots z_n$ the encoding of an integer $z < 2^n$ in $\mathtt{Gray}_n$ and by $\vec{z}[i, j], 1 \leq i \leq j \leq n$, the segment $z_i z_{i+1} \ldots z_j$ of $\vec{z}$. We define the *directional* $\mathtt{Gray}_n$ *encoding* of z as $\hat{z} = (\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_n)$, where for $1 \leq i \leq n$:

Procedure GrayTCAM($\mathcal{S}, n$)

1. $v := LCA(x', y)$; $\alpha = \Pi_L(u, v)$;

2. $\mathcal{R}_1 := \mathcal{C}_{n-1}(C_1)$;
   $\mathcal{R}_2 := \mathcal{C}_{n-|\alpha|-2}(C_2)$;
   $\mathcal{R}_3 := \mathcal{C}_{n-|\alpha|-1}(C_3)$;

3. **if** $(d_1 \leq d_2)$ **return** $(*\mathcal{R}_1 + 0\alpha*\mathcal{R}_2)$
   else return $(*\mathcal{R}_1 + 0\alpha\mathcal{R}_3)$

Figure 5.13: GrayTCAM($\mathcal{S}, n$): this procedure generates a minimal canonical TCAM representation for the range with the skeleton tree $\mathcal{S}$ of Figure 5.14 in $\texttt{Gray}_n$ encoding.

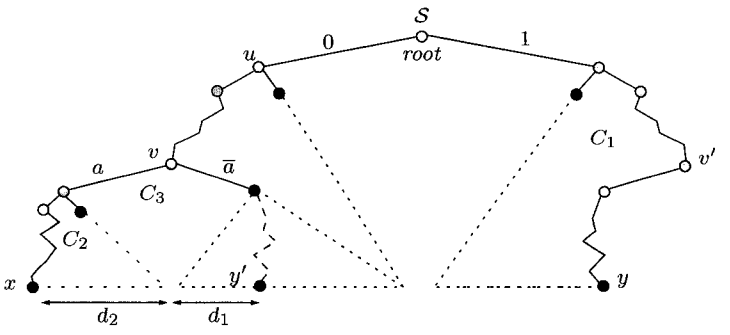


Figure 5.14: The structure of the skeleton tree $\mathcal{S}$ for a non-reciprocal range.

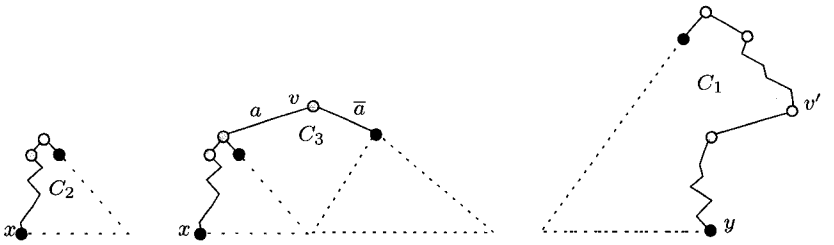


Figure 5.15: The computation of a minimal TCAM in Gray encoding for the skeleton tree of Figure 5.14 is reduced to the computation for the chain C_1 together with one of the chains C_2 or C_3.

$$\hat{z}_i = \begin{cases} L & \text{if } ((\hat{z}_{i-1}, z_i) = (L,0) \text{ or } (\hat{z}_{i-1}, z_i) = (R,1)) \\ R & \text{otherwise} \end{cases}$$

with $\hat{z}_0 \stackrel{\text{def}}{=} L$. The directional $\texttt{Gray}_n$ encoding of z is in fact a representation of the path corresponding to z on the dense tree $\texttt{Gray}_n$; for $1 \leq i \leq n$, $\hat{z}_i = L$ ($\hat{z}_i = R$) if the *ith* edge of this path goes to left (right).

For any two integer values x, y, the function $\theta(x,y)$ returns a triple (p,q,r) where $0 \leq p \leq q \leq r \leq n$ and p, q, and r are the first, second, and third occurrences of index i (minus 1) for which $\vec{x}_i \neq \vec{y}_i$; if $\vec{x}$ and $\vec{y}$ differ in less than three positions, then the corresponding value(s) of the triple are set to n. For instance if $\vec{x} = 00101001$ and $\vec{y} = 10111001$ then $\theta(x,y) = (0,3,8)$.

The algorithm for calculating the minimal canonical TCAM representation of a range $R = [x,y]$, where $y < 2^n$, is shown in Figure 5.16. In this algorithm, $\mathcal{L}(\vec{u})$, where $\vec{u}$ is the Gray encoding of an integer u on k bits, $1 \leq k \leq n$, denotes the minimal canonical TCAM for $[u, 2^k - 1]$ (i.e., a left-chain), which can be generated using the algorithm implied by Lemma 5.3.1.

In this way we arrive to our first main result.

ALGORITHM GrayTCAM($[x, y], n$)

1. $\theta(x, y) := (p, q, r)$;
2. if $(p = n)$ then **return** $(\vec{x})$;
3. if $(q = n)$ then **return** $(\vec{x}[1, p] * \mathcal{L}(\vec{x}[p + 2, n])$;
4. if $(\hat{x}_q = R)$ then **swap**(x, y);
5. if $(r = n$ **or** $\hat{x}_{r+1} = L)$ then
   return $\vec{x}[1, p]*\mathcal{L}(\vec{x}[p + 2, n]) + \vec{x}[1, q]*\mathcal{L}(\vec{x}[q + 2, n])$;
6. **return** $\vec{x}[1, p]*\mathcal{L}(\vec{x}[p + 2, n]) + \vec{x}[1, q]\mathcal{L}(\vec{x}[q + 1, n])$;

Figure 5.16: ALGORITHM GrayTCAM($[x, y], n$): generates a minimal canonical TCAM representation for the range $[x, y]$, $y < 2^n$, in the $\texttt{Gray}_n$ encoding.

Theorem 5.4.1. *The algorithm* ***GrayTCAM***$([x, y], n)$ *computes in time* $O(|\mathcal{R}|)$ *a minimal canonical TCAM* $\mathcal{R}$ *for a given range* $[x, y]$, $y < 2^n$, *in the reflected Gray encoding.*

5.4.2 Ranges in the lexicographic encoding

Let C_L, C_R be left and right-chains, respectively. Denote by $Merge(C_L, C_R)$ the skeleton tree which results by creating a new root and connecting C_L to the left child and C_R to the right child of the root (see $\mathcal{S}'$ in Figure 5.20).

The recursive algorithm $\texttt{LexTCAM}(\mathcal{S}, n)$ of Figure 5.17 returns a minimal canonical TCAM for a range represented by its skeleton tree $\mathcal{S}$ in the $\texttt{Lex}_n$ encoding. It employs a *top-down-reduction* approach to reduce the TCAM representation problem to another problem with a smaller skeleton tree or to a simple (terminal) case of a chain. The crucial notion in the algorithm is that of complementary chains (defined

in Section 3). If we *look* from the root v, the most challenging situation is when v has 4 grandchildren, as other cases are rather simple to manage. In this case, if C_L and C_R (i.e., the chains rooted at the leftmost and the rightmost grandchildren of v, respectively) are complementary, we use the construction of Figure 5.18 to reduce the problem to a smaller skeleton tree $\mathcal{S}'$.

Theorem 5.4.2. *The algorithm LexTCAM computes correctly in time $O(|\mathcal{R}|)$ a minimal canonical TCAM $\mathcal{R}$ for a range in the lexicographic encoding.*

The algorithm works obviously within the required complexities. The correctness follows from Lemma 5.3.4 and Propositions 5.4.1–5.4.4 which we present below.

The algorithm consists of 5 parts which correspond to the treatment of 5 different types of range skeleton trees described below (see also Figure 5.19):

ξ_0 – Chains;

ξ_1 – Double-chains whose roots have only one child;

ξ_2 – Double-chains whose roots have two children and two grandchildren;

ξ_3 – Double-chains whose roots have three grandchildren;

ξ_4 – Double-chains whose roots have four grandchildren.

We show that in each case the constructions of the algorithm are correct: the algorithm constructs a minimal canonical TCAM for $\mathcal{S}$ either by using Lemma 5.3.4 (cases 1 and 3) or from a solution for another range skeleton tree $\mathcal{S}'$ of strictly lower height (cases 2, 4, and 5). We first prove the following lemma.

Lemma 5.4.2. *Let I and I' be two ranges corresponding to the skeleton trees $\mathcal{S}$ and $\mathcal{S}'$ of Figure 5.20, respectively. A minimal TCAM for I has at least k rules more than a minimal TCAM for I', where $k \geq 1$ is the height difference between $\mathcal{S}$ and $\mathcal{S}'$.*

ALGORITHM LexTCAM($\mathcal{S}, n$)

1. if $\mathcal{S}$ is a chain then **return** $\mathcal{C}_n(\mathcal{S})$;
2. if $v = root(\mathcal{S})$ has one child z and the edge $v \rightarrow z$ has label $a \in \{0, 1\}$ then **return** a $LexTCAM(\mathcal{S}_z, n-1)$, where $\mathcal{S}_z$ is the tree rooted at z;
3. k := number of grandchildren of the root;
   let L, R be the leftmost and rightmost grandchildren of v, and C_L, C_R be the left and right-chains rooted in L and R, respectively;
   if $k = 2$ then **return** $01\mathcal{C}_{n-2}(C_L) + 10\mathcal{C}_{n-2}(C_R)$;
4. $\mathcal{S}'$:= $Merge(C_L, C_R)$; $\mathcal{R}'$:= $LexTCAM(\mathcal{S}', n-1)$;
   Split rules of $\mathcal{R}'$ w.r.t. the first symbol, i.e., $\mathcal{R}' = 0\mathcal{R}'_0 + 1\mathcal{R}'_1 + *\mathcal{R}'_*$;
   if $k = 3$ and L is a right child then
   return $01\mathcal{R}'_0 + 1*\mathcal{R}'_1 + *1\mathcal{R}'_* + 10** \cdots *$;
   if $k = 3$ and L is a left child then
   return $0*\mathcal{R}'_0 + 10\mathcal{R}'_1 + *0\mathcal{R}'_* + 01** \cdots *$;
5. $\mathcal{R}$:= $0*\mathcal{R}'_0 + *1\mathcal{R}'_1 + **\mathcal{R}'_* + 10** \cdots *$;
   if (C_L, C_R) are *complementary* then **return** $\mathcal{R}$
   else **return** $\mathcal{R} + 01** \cdots *$;

Figure 5.17: ALGORITHM LexTCAM($\mathcal{S}, n$): generates a minimal canonical TCAM representation for a range with the skeleton tree $\mathcal{S}$ in $\mathtt{Lex}_n$ encoding.

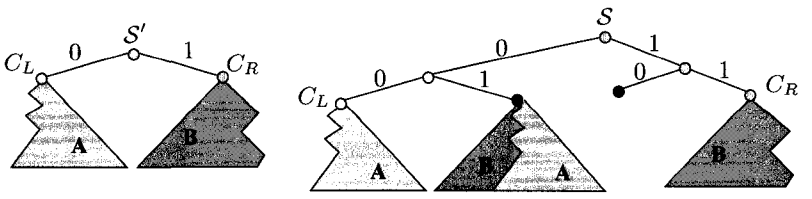


Figure 5.18: Skeletons trees $\mathcal{S}' = \mathit{Merge}(C_L, C_R)$ and $\mathcal{S}$ of Proposition 5.4.4 in case when C_L and C_R are complementary. Ranges corresponding to left-chain C_L and right-chain C_R are represented by trees A and B, respectively.

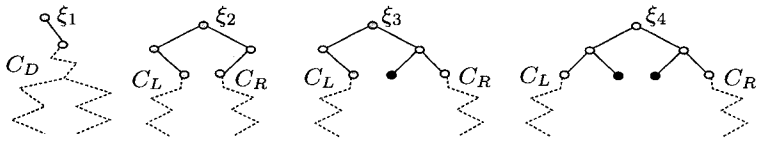


Figure 5.19: Double-chains ξ_1, ξ_2, ξ_3, and ξ_4. The double-chain C_D, left-chain C_L, and right-chain C_R are non-empty. Notice that in some cases the chains cannot be trivial (i.e., a single node tree). For example, in ξ_4 both chains C_L and C_R are non-trivial.

Proof. Let $\mathcal{R}$ be a minimal canonical TCAM for I and $\mathcal{R}_C$ be the set of those rules in $\mathcal{R}$ which enter into the chains C_L and C_R (i.e., rules that intersect with $0^k1{*}{*}\cdots{*}$ or with $1^{k+1}{*}{*}\cdots{*}$). The rules of $\mathcal{R}_C$ can be turned into a TCAM for I' by replacing $k+1$ initial symbols of each rule with one of 0, 1, or $*$.

If $k = 1$, then the I-essential rule $10{*}{*}\cdots{*}$ cannot be covered by other I-essential rules. Therefore a minimal TCAM for I has exactly one rule more than a minimal TCAM for I'. For $k > 1$, since C_R is not a single node tree (i.e., at least $111\cdots1 \notin C_R$), no inner node of $\mathcal{S}$ at level k, except the parents of C_L and C_R, can be covered by rules from $\mathcal{R}_C$. More precisely, no string which has both a 0 and a 1 in its k first bits, 0 at position $k+1$, and then only 1 in all positions bigger than $k+1$, can be covered by rules from $\mathcal{R}_C$. Thus, by proof of Lemma 5.3.5, we need at least k rules outside of $\mathcal{R}_C$ to cover all those k-level inner nodes. □

Step 1. For skeleton trees of type ξ_0 we directly apply Lemma 5.3.4.

Step 2. The case of a skeleton tree of type ξ_1 is simple since there is the one-to-one correspondence between all TCAM representations of (the range of) $\mathcal{S}$ and all TCAM representations of (the range of) $a^{-1}\mathcal{S}$, where a is the label of the unique edge of the root of $\mathcal{S}$. More formally:

Proposition 5.4.1 (Skeleton trees of type ξ_1). *Let $\mathcal{S}$ be a range skeleton tree in* $\mathtt{Lex}_n$ *with $h(\mathcal{S}) \leq n$, and for some $a \in \{0,1\}$, $a^{-1}\mathcal{S} = \mathcal{S}' \neq T_\emptyset$ and $\overline{a}^{-1}\mathcal{S} = T_\emptyset$. $\mathcal{R}$ is a minimal canonical TCAM for $\mathcal{S}'$ encoded on $n-1$ bits if and only if $a\mathcal{R}$ is a minimal canonical TCAM for $\mathcal{S}$ on n bits.*

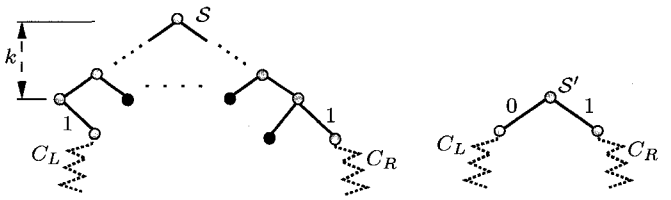


Figure 5.20: Two skeleton trees $\mathcal{S}$ and $\mathcal{S}'$ from Lemma 5.4.2. C_L and C_R are left and right-chains, respectively. Since $\mathcal{S}$ is a skeleton tree, C_R cannot be a single node tree.

Step 3. For the skeleton trees of type ξ_2 we have:

Proposition 5.4.2 (Skeleton trees of type ξ_2). *Let $\mathcal{S}$ be a range skeleton tree in* $\mathtt{Lex}_n$ *such that $(00)^{-1}\mathcal{S} = (11)^{-1}\mathcal{S} = T_\emptyset$, $(01)^{-1}\mathcal{S} = C_L \neq T_\emptyset$, and $(10)^{-1}\mathcal{S} = C_R \neq T_\emptyset$, with $n \geq h(\mathcal{S})$. If $\mathcal{R}_L$ is a minimal canonical TCAM for C_L on $n-2$ bits and $\mathcal{R}_R$ is a minimal canonical TCAM for C_R on $n-2$ bits, then $\mathcal{R} = 01\mathcal{R}_L + 10\mathcal{R}_R$ is a minimal canonical TCAM for $\mathcal{S}$ encoded on n bits.*

Proof. Every rule in a TCAM for $\mathcal{S}$ starts by 01 or 10. Therefore, any TCAM covering $\mathcal{S}$ can be written as $01\mathcal{R}_L + 10\mathcal{R}_R$, where $\mathcal{R}_L$ and $\mathcal{R}_R$ cover C_L and C_R, respectively. □

Step 4. If a skeleton tree $\mathcal{S}$ is of type ξ_3 then either $(00)^{-1}\mathcal{S} = T_\emptyset$ or $(11)^{-1}\mathcal{S} = T_\emptyset$. Since these two cases are very similar, in the following proposition we consider only one case, $(00)^{-1}\mathcal{S} = T_\emptyset$, which corresponds to the graphical representation of ξ_3 in Figure 5.19.

Proposition 5.4.3 (Skeleton trees of type ξ_3). *Let $\mathcal{S}$ be a range skeleton tree in* $\mathtt{Lex}_n$ *such that $(00)^{-1}\mathcal{S} = T_\emptyset$, $(01)^{-1}\mathcal{S} = C_L \neq T_\emptyset$, $(10)^{-1}\mathcal{S} = \bullet$, and $(11)^{-1}\mathcal{S} = C_R \neq T_\emptyset$, with $n \geq h(\mathcal{S})$. Also let $\mathcal{S}'$ be a range skeleton tree in $\mathtt{Lex}_{n-1}$ such that $0^{-1}\mathcal{S}' = C_L$ and $1^{-1}\mathcal{S}' = C_R$. If $\mathcal{R}' = 0\mathcal{R}'_0 + 1\mathcal{R}'_1 + *\mathcal{R}'_*$ is a minimal canonical TCAM for $\mathcal{S}'$, then*

$$\mathcal{R} = 01\mathcal{R}'_0 + 1*\mathcal{R}'_1 + *1\mathcal{R}'_* + 10(*^{n-2})$$

is a minimal canonical TCAM for $\mathcal{S}$.

Proof. By Lemma 5.4.2, with $k = 1$. □

Step 5. In case of skeleton trees of type ξ_4, i.e., when all four grandchildren of the root are non-empty, we distinguish two cases; the chains rooted in the leftmost grandchild and the rightmost grandchild (i.e., chains C_L and C_R in ξ_4 of Figure 5.19) are *complementary* or are not.

Proposition 5.4.4 (Skeleton trees of type ξ_4). *Let $\mathcal{S}$ be a range skeleton tree in* $\mathtt{Lex}_n$ *such that $(00)^{-1}\mathcal{S} = C_L \neq T_\emptyset$, $(01)^{-1}\mathcal{S} = \bullet$, $(10)^{-1}\mathcal{S} = \bullet$, and $(11)^{-1}\mathcal{S} = C_R \neq T_\emptyset$, with $n \geq h(\mathcal{S})$. Also suppose that $\mathcal{S}'$ is a range skeleton tree in $\mathtt{Lex}_{n-1}$ with a minimal canonical TCAM $\mathcal{R}' = 0\mathcal{R}'_0 + 1\mathcal{R}'_1 + *\mathcal{R}'_*$, such that $0^{-1}\mathcal{S}' = C_L$ and $1^{-1}\mathcal{S}' = C_R$.*

1. *If C_L and C_R are complementary chains then:*

$$\mathcal{R} = 0*\mathcal{R}'_0 + *1\mathcal{R}'_1 + **\mathcal{R}'_* + 10(*^{n-2})$$

is a minimal canonical TCAM for $\mathcal{S}$.

2. Otherwise a minimal canonical TCAM for $\mathcal{S}$ is:

$$\mathcal{R} = 0*\mathcal{R}'_0 + *1\mathcal{R}'_1 + **\mathcal{R}'_* + 10(*^{n-2}) + 01(*^{n-2})$$

Proof. If C_L and C_R are complementary, then the construction in Figure 5.18 shows that TCAM $\mathcal{R} = 0*\mathcal{R}'_0 + *1\mathcal{R}'_1 + **\mathcal{R}'_* + 10(*^{n-2})$ covers $\mathcal{S}$, whenever $0\mathcal{R}'_0 + 1\mathcal{R}'_1 + *\mathcal{R}'_*$ covers $\mathcal{S}'$ and the number of rules in $\mathcal{R}$ is one more than that of $\mathcal{R}'$. Besides Lemma 5.4.2 guarantees that no smaller TCAM can cover $\mathcal{S}$. Notice that another symmetric construction for $\mathcal{R}$ is also possible: $\mathcal{R} = *0\mathcal{R}'_0 + 1*\mathcal{R}'_1 + **\mathcal{R}'_* + 01(*^{n-2})$.

If C_L and C_R are non-complementary, then $01*\cdots*$ (resp., $10*\cdots*$) is an $\mathcal{S}$-essential rule which cannot be covered by other $\mathcal{S}$-essential rules. Therefore, $10*\cdots*$ and $01*\cdots*$ have to be in any minimal canonical solution for $\mathcal{S}$. Notice that in case when C_L and C_R are non-complementary, there are four variants for construction of a minimal canonical TCAM for $\mathcal{S}$ from a minimal canonical TCAM for $\mathcal{S}'$:

$$\mathcal{R} = \alpha\mathcal{R}'_0 + \beta\mathcal{R}'_1 + **\mathcal{R}'_* + 10(*^{n-2}) + 01(*^{n-2}),$$

for $\alpha \in \{*0, 0*\}$ and $\beta \in \{*1, 1*\}$. □

5.5 Simulation Results

Our algorithm for generating a minimal TCAM representation of a given range under the lexicographic encoding, enables us for the first time to determine how far the prefix-based method is from the optimal solution. While in Lemma 5.3.5, we showed that there are ranges for which the number of rules generated by the prefix-based method is twice larger than that of the optimal solution for the lexicographic encoding, we still do not know the difference in the average performance of the two algorithms.

We start our simulations in the context of today's multifield packet classification rules. Normally, each IPv4 Access Control List (ACL) rule consists of five fields

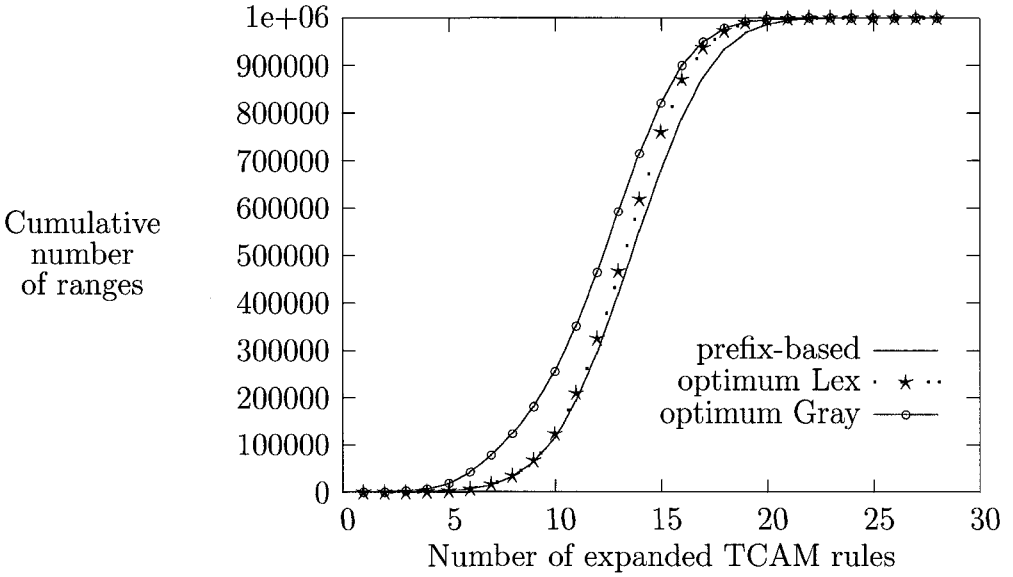


Figure 5.21: Cumulative distribution of expansion rates for a set of 10^6 random ranges over $[0, 2^{16} - 1]$.

as follows: source address (32 bits), destination address (32 bits), source port (32 bits), destination port (32 bits), and protocol (8 bits). Since the width of the two fields that may include ranges (i.e., source port and destination port) is 16 bits, the encoding width in our first simulations is 16 bits. For a set of 10^6 random ranges over the domain $D = [0, 2^{16} - 1]$, Figure 5.21 shows the cumulative distribution of the number of ranges that expand to a given number of TCAM rules for the prefix-based method, and the optimal solutions for the lexicographical and Gray encodings. It can be deducted from these curves that the average expansion rate of the prefix-based method, the optimal solutions for the lexicographic encoding, and the optimal solution for the Gray encoding are 14.1, 13.6, and 12.5, respectively.

Figure 5.22 illustrates the cumulative distribution of the number of ranges that expand to a given number of TCAM rules for the same three methods and a set of 10^6 random ranges over the domain $D = [0, 2^{32} - 1]$. In this case the average expansion

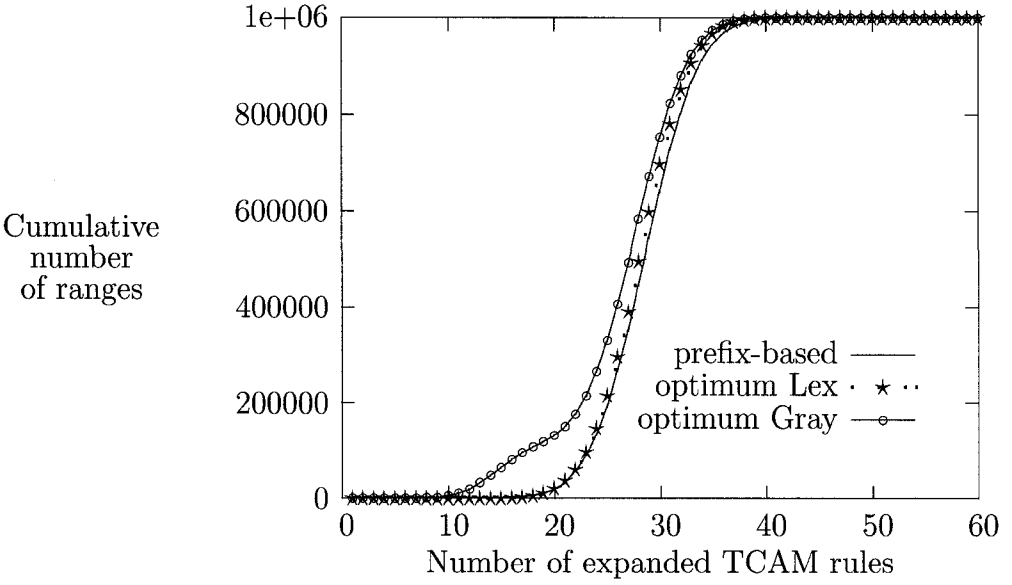


Figure 5.22: Cumulative distribution of expansion rates for a set of 10^6 random ranges over $[0, 2^{32} - 1]$.

rates are 29, 28.5, and 26.7 for the prefix-based method, the optimum solution for the lexicographic encoding, and the optimum solution for the Gray encoding, respectively.

The first result of these simulations is that the average-case performance of prefix-based method is much better than its worst-case performance. Beside, this average performance is not more than 0.5 and 2 rules far from the average performance of the optimal solutions for the lexicographic and Gray encodings, respectively. However, as it has been observed in [65], in the past few years the percentage of ACL rules with two range fields has had a fast increasing trend. Since the expansion rate for any such ACL rule is equal to the multiplication of the expansion rates of its range fields, our algorithms can provide a considerable gain in the amount of TCAM required for today's industrial solutions to multifield packet classification.

5.6 Conclusions

There has been a continuous trend in the increase of the number of rules, and also the percentage of rules with range fields, in the policy databases that are used for multifield packet classification. On the other hand, some range fields are not efficiently represented in TCAM by the prefix-based expansion method which is still widely used in the available industrial solutions for multifield packet classification. In this chapter we presented a systematic approach for exploring this problem. We showed that replacing the lexicographic encoding of the prefix-based method with the binary reflected Gray encoding provides us with a gain in the expansion rate. Also for a family of minimum-width binary encoding schemes, which include the lexicographic encoding and the binary reflected Gray encoding, we provided a tight lower bound on the worst-case expansion rate of the ranges in TCAM.

Finally we presented two fast and simple algorithms that generate a minimal TCAM representation for a given range: one algorithm for the binary reflected Gray encoding and the other for the lexicographic encoding. Being as simple and as fast as the prefix-based method, these very efficient algorithms can replace the prefix-based algorithm in network packet processing applications such as the ACL.

Chapter 6

Concluding Remarks

This thesis was concerned with different aspects of devising an efficient architecture that enables us to implement high-performance FSTs in hardware. The target application of high-performance FSTs was to provide a solution for the problem of content inspection. However, the results can be used for any application that needs a hardware-implemented high-performance FST. In this chapter, we briefly review the contributions of the thesis and suggest some directions for future research work.

6.1 Summary of Contributions

The problem of content inspection can be represented and solved by finite state machines. Nevertheless, as any other hardware-based solution to this problem, a state-based architecture should address two major challenges: it needs to meet today's multi gigabit rate processing speed requirements and besides, it should be able to represent the rules of inspection policy in the limited memory of hardware. In this thesis, we first resolved the speed issue by using high-performance transducers instead of normal transducers. A high-performance FST potentially provides a real-time solution to the problem of content inspection at any desired speed by processing a variable number of bits at each of its states. We then addressed the memory issue by

introducing the TLT architecture for hardware implementation of high-performance FSTs. The TLT architecture has a two-level memory structure which enables it to implement high-performance FSTs without suffering from the memory explosion problem associated to the traditional look-up table implementation of transducers.

An appropriate medium for storing the layout entries of the TLT architecture are TCAMs whose parallel structure provides us a considerable boost in the inspection speed. We proved that the problem of representing a given layout entry in TCAM with minimum number of rules is an NP-hard problem and then devised dynamic programming techniques for generating TLTs with reduced TCAM requirements. Our simulation results confirm that the TLT architecture implements high-performance FSTs efficiently. The results from this part of the work have been presented in [28].

The speed evaluation of CIEs has remained a largely untouched problem in the computer networking community. In practice, due to the inclusion of the payload in the processing of the packets, the amount of time spent on processing each packet depends mainly on its contents and not on its length. As such traditional metrics of packet processing speed, whose main assumption is that each packet has constant (or at most well-bounded) per-packet component (e.g., multi-field packet classification), or it has a flat per-byte cost (e.g., packet reception and storage, calculation of checksums) cannot be applied to the content inspection processing. As the second contribution of this thesis, we proposed *worst-case throughput* as a new criterion for speed evaluation of CIEs. We also explained how we can express and calculate the worst-case throughput of a CIE in terms of the parameters of a graphical model of the CIE. The results of this part were presented in [29].

The third contribution of this thesis was to devise a new algorithm for calculating the worst-case throughput of CTs. The functionality of CTs can be explained by an associated stack. This stack enables CTs to solve content inspection problems that deal with packets whose contents are from a subclass of context-free languages

which includes non-regular languages such as the simplified XML language. We have presented the results from this part of the work in [25] and [26].

As the forth contribution of the thesis we provided a formal framework for exploring the problem of range representation in TCAM. This problem has recently started to be investigated specially due to its importance in industrial solutions for the problem of multifield packet classification where range fields are represented by TCAM rules. However, none of the existing solutions has directly improved the traditional prefix-expansion method which is still the most widely used method in practice. We focused on the range-expansion with minimum-width TCAM and provided a tight lower bound on the worst-case expansion rate of TCAM rules for a class of encoding schemes which includes the standard binary encoding and the binary reflected Gray encoding. We have presented the results of this part in [27]. Finally, we presented two fast and simple algorithms that generate a minimal TCAM representation for a given range in the binary reflected Gray encoding and the lexicographic encoding. These two efficient algorithms are as simple as the prefix-based algorithm and thus provide an immediate performance improvement in network packet processing applications such as the Access Control List (ACL) implementation.

6.2 Future Work

While our simulation results confirm that the TLT architecture of Chapter 3 is an efficient method of implementing high-performance FSTs in hardware, this process is still far from optimal. There are some aspects of this architecture and some trade offs that can be investigated in the future. For instance, in Chapter 3 we proved that for a given state of a high-performance FST, the problem of encoding the incoming string labels of its outgoing edges with a minimum number of TCAM rules is an NP-hard problem. As a consequence of this fact, we have relied on a heuristic to encode the

labels of outgoing edges of a state in TCAM. A possible future work is to look for other heuristics and approaches that can help us to represent the labels with a smaller number of TCAM rules. For this purpose, one can explore the techniques that have been already developed for the related problem of two-level logic minimization [69].

Another possible future work is to explore whether by using the so called "failure" transitions of the classic Aho-Corasick algorithm [70], we can reduce the TCAM requirements of implementing HPTs with the TLT architecture. This idea looks specially promising if we want to implement a HPT which is obtained from a multiple-string matching problem.

In Chapter 5, we developed two linear-time algorithms that provide a minimum TCAM representation of a given range for the lexicographic and the Gray encodings. As a possible future direction, we can investigate whether these algorithms can be used to provide an efficient solution to the more general problem of representing an arbitrary subset of a domain in TCAM.

Finally, as our transducer-based approach is not the only way of solving the content inspection problem, it is still interesting to combine it with other existing solutions for this problem.

Bibliography

[1] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "Pathfinder: A pattern-based packet classifier," in *Operating Systems Design and Implementation*, pp. 115–123, 1994.

[2] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, "Fast and scalable layer four switching," in *Proceedings of ACM Sigcomm*, vol. 28, pp. 191–202, October 1998.

[3] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *SIGCOMM*, pp. 203–214, 1998.

[4] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *ACM Sigcomm*, pp. 147–160, 1999.

[5] B. Haagdorens, T. Vermeiren, and M. Goossens, "Improving the performance of signature-based network intrusion detection sensors by multi-threading.," in *WISA*, pp. 188–203, 2004.

[6] A. Kuzmanovic and E. W. Knightly, "Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 75–86, ACM Press, 2003.

[7] P. J. de Villiers and R. A. Botha, "An architecture for selling xml content," in *Workshop on Emerging Applications for Wireless and Mobile access*, 2003.

[8] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet Computing*, vol. 3, no. 3, pp. 28–39, 1999.

[9] J. Coit, S. Staniford, and J. McAlerney, "Towards faster string matching for intrusion detection or exceeding the speed of snort," in *DARPA Information Survivability Conference*, vol. 1, pp. 367–373, June 2001.

[10] M. Karagiannis, "How to create a Stealth Packet Scrubber using Hogwash." Application notes for Hogwash, 2001.

[11] F. Baboescu and G. Varghese, "Scalable packet classification.," in *SIGCOMM*, pp. 199–210, 2001.

[12] M. Roesch, "Snort-lightweight intrusion detection for netwroks," in *USENIX LISA'99 conference*, Nov. 99.

[13] L. Schaelicke, T. Slabach, B. Moore, and C. Freeland, "Characterizing the performance of network intrusion detection sensors," in *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, Lecture Notes in Computer Science, (Berlin–Heidelberg–New York), Springer-Verlag, September 2003.

[14] R. Kempke and A. McAuley, "Ternary cam memory architecture and methodology," August 1996. US Patent 5,841,874.

[15] F. Inc., "Addressing the limitations of deep packet inspection with complete content protection." White paper, January 2004.

[16] F. Inc., "The importance of web content filtering." White paper, 2004.

[17] A.-R. I. C. Inspection and M. Solution, "Content inspection in high capacity networks." White paper.

[18] B. Technologies, "Deep content inspection: Beyond deep packet inspection." White paper, November 2003.

[19] L. Wirbel, "How deep is my packet?," *Network Magazine*, May 2004.

[20] Y. H. Cho, S. Navab, and W. H. Mangione-Smith, "Specialized hardware for deep network packet filtering," in *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, (London, UK), pp. 452–461, Springer-Verlag, 2002.

[21] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel Bloom filters," in *Hot Interconnects*, (Stanford, CA), pp. 44–51, Aug. 2003.

[22] Y. Fang, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using tcam.," in *ICNP*, pp. 174–183, 2004.

[23] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection.," in *INFOCOM*, vol. 4, pp. 2628–2639, March 2004.

[24] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[25] J. Czyzowicz, W. Fraczak, and M. Yazdani, "Throughput of high-performance concatenation state machines," in *Proceedings of the sixteenth australasian workshop on combinatorial algorithms (AWOCA2005)*, (Ballarat, Australia), pp. 85–94, September 2005.

[26] J. Czyzowicz, W. Fraczak, and M. Yazdani, "Computing the throughput of concatenation state machines," *Elsevier Journal of Discrete Algorithms*, vol. 6, pp. 28–36, March 2008.

[27] W. Fraczak, W. Rytter, and M. Yazdani, "Tcam representations of intervals of integers encoded by binary trees," in *Proceedings of IWOCA Conference*, (Lake Macquarie, Newcastle, Australia), November 2007.

[28] M. Yazdani, W. Fraczak, F. Welfeld, and I. Lambadaris, "Two level state machine architecture for content inspection engines," in *Proceedings of IEEE INFOCOM 2006*, pp. 1–12, 2006.

[29] M. Yazdani, W. Fraczak, F. Welfeld, and I. Lambadaris, "A criterion for speed evaluation of content inspection engines," in *Fifth International Conference on Networking (ICN 2006)*, pp. 19–24, IEEE Computer Society, 2006.

[30] T. L. Booth, *Sequential Machines and Automata Theory.* New York: J. Wiley & Sons, 1967.

[31] J. Carroll and D. Long, *Theory of finite automata with an introduction to formal languages.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[32] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 2001.

[33] D. C. Kozen and D. C. Kozen, *Automata and Computability.* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.

[34] H. J. E., "An *nlogn* algorithm for minimizing states in a finite automaton,"

[35] S. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies* (C. Shannon and J. McCarthy, eds.), vol. 34 of *Annals of Math. Studies*, pp. 3–42, Pinceton University Press, 1956.

[36] J. Levin, T. Mason, and S. Brown, *Lex and Yaac.* O'Reilly & Associates, 2nd ed., October 1992.

[37] H. Hahn, *The Unix Companion.* Mcgraw-Hill Osborne Media, September 1995.

[38] R. Schwartz and T. Phoenix, *Learning Perl.* O'Reilly & Associates, 3rd ed., July 2001.

[39] M. Nossik, "Hardware holds value for classifying packets," *EE Times*, November 2003.

[40] F. Risso and M. Baldi, "Netpdl: An extensible xml-based language for packet header description.," *Computer Networks*, vol. 50, no. 5, pp. 688–706, 2006.

[41] J. Gimpel, "A method of producing a boolean function having an arbitrarily prescribed prime implicant table," *IEEE Trans. Computers*, vol. 14, pp. 485–488, 1965.

[42] J. Cochet-Terrasson, G. Cohen, S. Gaubert, M. M. Gettrick, and J.-P. Quadrat, "Numerical computation of spectral elements in max-plus algebra," in *Proceeding IFAC Conference on Systems Structure and Control*, 1998.

[43] A. Dasdan, S. Irani, and R. Gupta, "An experimental study of minimum mean cycle algorithms," tech. rep., Univ. of California, Irvine, July 1998.

[44] G. Koelemeijer, "The power and Howard algorithms in the (max,+) semiring," in *Proc. of the Fifth Workshop on Discrete Event Systems (WODES00)*, (Ghent, Belgium), IEE, 2000.

[45] E. Lawler, *Combinatorial optimization: networks and matroids.* New York, NY, USA: Holt, Reinhart, and Winston, 1976.

[46] T. Kohonen, *Content-Addressable Memories.* Springer-Verlag, New York, 1980.

[47] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, pp. 712–727, March 2006.

[48] "Intel case study: IDT PAX.Port 2500 Content Inspection Engine (CIE) and Intel IXP2400 network processor." Online publication, 2004. `http://www.intel.com/design/network/casestudies/idt_04.pdf`.

[49] M. Hartmann and J. Orlin, "Finding minimum cost to time ratio cycles with small integral transit times," *Networks*, vol. 23, pp. 567–74, 1993.

[50] F. Bacelli, G. Cohen, G. Olsder, and J. Quadrat, *Synchronization and Linearity.* New York, NY, USA: John Wiley & Sons, 1992.

[51] S. Burns, *Performance analysis and optimization of asynchronous systems.* PhD thesis, California Institute of Technology, 1991.

[52] J. Teich, S. Sriram, L. Thiele, and M. Martin, "Performance analysis and optimization of mixed asynchronous synchronous systems," *IEEE Transaction on Computer-Aided Design*, vol. 16, no. 5, pp. 973–84, 1997.

[53] K. Ito and K. Parhi, "Determining the minimum iteration period of an algorithm," *J. VLSI Signal Processing*, vol. 11, pp. 229–244, Dec. 1995.

[54] "Snort system." www.snort.org.

[55] W. Debski and W. Fraczak, "Concatenation state machines and simple functions," in *Implementation and Application of Automata, CIAA 2004*, vol. 3317 of *LNCS*, pp. 113–124, Springer, 2004.

[56] W. Fraczak and A. Podolak, "A characterization of s-languages," *Information Processing Letters*, vol. 89, no. 2, pp. 65–70, 2004.

[57] A. J. Korenjak and J. E. Hopcroft, "Simple deterministic languages," in *Proc. IEEE 7th Annual Symposium on Switching and Automata Theory*, IEEE Symposium on Foundations of Computer Science, pp. 36–46, 1966.

[58] J.-M. Autebert, J. Berstel, and L. Boasson, "Context-free languages and pushdown automata," in *Handbook of Formal Languages* (A. Salomaa and G. Rozenberg, eds.), vol. 1, Word Language Grammar, pp. 111–174, Berlin: Springer-Verlag, 1997.

[59] D. Caucal, J. Czyzowicz, W. Fraczak, and W. Rytter, "Efficient computation of throughput values of context-free languages," in *CIAA*, pp. 203–213, 2007.

[60] G. Levi and F. Sirovich, "Generalized and/or graphs," *Artificial Intelligence*, vol. 7, pp. 243–259, 1976.

[61] A. Martelli and U. Montanari, "Additive and/or graphs," in *Proc. IJCAI*, vol. 3, pp. 1–11, 1973.

[62] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to cams?," in *INFOCOM*, vol. 1, pp. 53–63, 2003.

[63] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting.," in *SIGCOMM*, pp. 213–224, 2003.

[64] D. Taylor and J. Turner, "Scalable packet classification using distributed crossproducting of field labels," in *Proc. of IEEE Infocom*, vol. 1, pp. 269–280, March 2005.

[65] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary cams.," in *SIGCOMM*, pp. 193–204, 2005.

[66] H. Liu, "Efficient mapping of range classifier into ternary-cam," in *HOTI '02: Proceedings of the 10th Symposium on High Performance Interconnects HOT Interconnects (HotI'02)*, (Washington, DC, USA), p. 95, IEEE Computer Society, 2002.

[67] E. Gilbert, "Gray codes and paths on the n-cube," *Bell Systems Technical Journal*, vol. 37, pp. 815–826, 1958.

[68] F. Gray, "Pulse code communications," March 1953. US Patent 2,632,058.

[69] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[70] A. Aho and M. Corasick, "Efficient string matching: and aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.