國立臺灣大學電機資訊學院電機工程研究所

博士論文

Graduate Institute of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Doctoral Dissertation

具多字元狀態轉移之高效字串比對引擎

Efficient String Matching Engines with Multiple-Character State Transitions

陳建吉

Chien-Chi Chen

指導教授：王勝德 教授

Advisor: Sheng-De Wang, Ph.D.

中華民國 103 年 6 月

June 2014

國立臺灣大學博士學位論文
口試委員會審定書

具多字元狀態轉移之高效字串比對引擎
Efficient String Matching Engines with Multiple-Character State Transitions

本論文係陳建吉君（學號 D93921012）在國立臺灣大學電機工程學系完成之博士學位論文，於民國 103 年 7 月 22 日承下列考試委員審查通過及口試及格，特此證明。

口試委員：　王勝德　（簽名）
（指導教授）

雷欽隆　　陳銘憲

曾俊元　　顏嗣鈞

系主任　[illegible]　（簽名）

致謝

十年磨一劍，很高興能順利完成學位。首先，最感謝的當然是我的指導教授，王勝德老師，感謝他的耐心與支持，我才得以完成學位。另外，在這些年中，許多來來去去的學長與學弟妹們，互相的提攜與打氣，也是我前行的力量之一。進入博士班的同時，也是我的生活變化的開始。首先，身分變成已婚單身的狀態，然後陸續接觸一些成長課程，包括兩次的單車環島，想法和生活慢慢地變得更為自在和積極。與我成長的同時，三個男孩也一起成長，眞心感謝他們的獨立，讓我能專注於自己的事。最常被問的是，為何要花這麼多時間在這學位上，其實在回到學生身分，專注於求學的過程當中，就是一種成就吧。隨著學業的完成，我的十年大運將進入比較具有挑戰的階段，也意味著，我的另一階段的生活體驗即將開始。

感謝老天，謝謝所有，出現在我生命中的貴人。

中文摘要

由 Aho 和 Corasick 所提出的演算法 (簡稱 AC 演算法) 可以很有效率地在一段文字中搜尋多個關鍵字所在的位置，因而被廣泛地使用於完全字串比對。然而，AC 演算法在運作時，只能一次處理一個字元，因而其效能受限於運作時脈。本論文依據 AC 演算法，提出新穎的具有多字元狀態轉移之字串比對架構，可以平行檢視多個字元，藉以加倍提升字串比對的效能。首先，說明實作 AC 演算法的三種有限自動機（finite automaton，簡稱 FA）的作法，包括確定性有限自動機 (Deterministic Finite Automaton，簡稱 DFA)、非確定有限自動機 (Nondeterministic Finite Automaton，簡稱 NFA)、以及複合有限自動機 (Hybrid Finite Automaton，簡稱 hybrid FA) 的作法。接著，提出一種推導演算法，藉以將前述 FA 推導爲多字元的 FA，使其能夠平行檢視多個字元。同時，因爲平行檢視多個字元所引起的對齊問題 (alignment problem)，也藉由輔助轉移函式 (assistant transition) 來解決。所推導的多字元 FA 使用可程式元件來評估，所得到的評估結果顯示其能有效地加倍提升字串比對的效能。其中所得到的最佳結果爲 16字元 AC-NFA 的實作，其可運作於 167.36MHz 的時脈，換算得到的字串比對效能爲 21.4Gbps。

關鍵字：字串比對，確定性及非確定有限自動機，入侵偵測系統

Abstract

The algorithm of Aho and Corasick (AC-algorithm) can locate multiple keywords in a text efficiently; and thus it is widely used in the exact string matching. However, the AC-algorithm processes the text character by character with a performance limited by the processing clock. This thesis proposes novel string-matching architectures with multiple-character state transitions based on the AC-algorithm, which can inspect multiple characters in parallel to multiply the throughput. At first, three finite automaton (FA) approaches including Deterministic Finite Automaton (DFA), Nondeterministic Finite Automaton (NFA), and hybrid FA approaches are presented to implement the AC-algorithm. Subsequently, a derivation algorithm is proposed to derive these FAs to multi-character FAs to allow for the inspection of multiple characters in parallel. Additionally, the alignment problem, which occurs while multiple characters are being inspected in parallel, is solved by using assistant transitions. The derived multi-character FAs are evaluated on programmable devices and the evaluation results indicate that the throughput can be multiplied effectively. The achievable throughput of the best result is 21.4Gbps obtained by a 16-character AC-NFA implementation operating at a 167.36MHz clock.

Keywords: String-matching, deterministic and nondeterministic finite automaton, intrusion detection system

Contents

List of Figures

List of Tables

Chapter 1

Introduction

String matching plays a major role in many applications, such as network intrusion detection systems (NIDS) and bioinformatics. String matching generally includes exact string matching and regular expression matching; exact string matching is more efficient yet less flexible for searching keywords in a text than regular expression matching. This thesis mainly focuses on developing efficient exact string matching engines.

1.1 Exact String Matching and Motivations

Aho and Corasick have proposed an efficient algorithm (AC-algorithm) for exact string matching that can locate all occurrences of multiple keywords in a text in a one pass search [1]. Some applications like NIDS that need to inspect the data stream on line usually first use exact string matching to find suspected data quickly, then check the results further by other more complicated and slower approaches, such as regular expression matching.

The network bandwidth has ever been increasing as the advances of communication and integrated circuit technology. The throughput of packet inspection for NIDS must also be sped up to keep up with the network throughput otherwise the packet inspection will become the bottleneck of the network security check. A hardware string matching implementation can effectively accelerate the packet inspection process. In early work, most hardware implementations of the string matching inspect the packet data character by character, and the throughput of string matching is limited by the achievable clock rate of the hardware [2, 3]. Recently, some hardware string matching approaches attempted to inspect multiple characters in a single cycle to further improve the throughput of packet inspection [4–7].

Although some researches attempted to develop a string matching engine capable of inspecting multiple characters in parallel, a systematic approach for developing such a string matching engine has yet to be developed. Furthermore, most of the researches developed the architecture according to the properties of keyword sets to optimize the performance, which lack flexibility that different keyword sets can be processed by simply updating the setting data. Therefore, this work intends to develop an intuitive and systematic derivation approach to derive multi-character transitions from an original finite state machine (FSM). Moreover, various architectures are proposed to implement the derived multi-character transitions. A generalized multi-character string matching architecture is also proposed, which can process different keyword sets by simply updating the configurations.

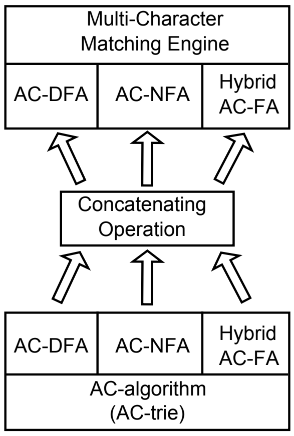


Figure 1.1: Overview of the proposed work

1.2 Approach Overview

This thesis presents approaches based on the AC-algorithm to enhance the performance of exact string matching. The overview of this research can be illustrated as Fig. 1.1. A prefix-tree created according to the AC-algorithm is known as an AC-trie, which consists of goto functions and failure functions. All the occurrences of keywords can be located in a one pass search through the goto functions and the failure functions in the matching process. However, the failure functions in an AC-trie causing multiple state transitions in a matching cycle make it inconvenient to implement the AC-algorithm in a deterministic hardware circuit. A general approach is converting an AC-trie to a Deterministic Finite Automaton (DFA), called

AC-DFA, or a Nondeterministic Finite Automaton (NFA), called AC-NFA, to eliminate the failure functions and then implementing the derived AC-DFA or AC-NFA in a hardware circuit. The advantage of the AC-DFA approach is that it can be implemented as a general architecture, while its disadvantage is space expensive. In contrast, the advantage of the AC-NFA approach is space efficient, while its disadvantage is that the hardware implementation is dependent on the keyword set. This thesis proposes a hybrid finite automaton, called hybrid AC-FA, that combine the AC-DFA and AC-NFA to combine both the advantages of DFA and NFA.

However, the string matching engine inspects a text character by character with a performance limited by the achievable maximum operating clock; therefore it needs to inspect multiple characters in parallel to further increase the performance. While multiple characters are being inspected in parallel, besides the complexity is increased, a common problem has to be considered is the alignment problem of starting and ending characters. The alignment problem includes two cases: a pattern does not begin at the first character of the inspecting characters, and a pattern does not end at the last character of the inspecting characters. Consider the inspecting characters ‘they’ as an example, in which the pattern ‘he’ begins at the second character and ends at the third character. In this scenario, neither the beginning of the pattern aligns with the first character of the inspecting characters nor the ending of the pattern aligns with the last character of the inspecting characters.

This thesis develops an intuitive algorithm to derive multi-character transitions from an AC-DFA, an AC-NFA, or a hybrid AC-FA, where each transition can match multiple characters at a time. This derivation algorithm also includes using assistant transitions and a pseudo state to resolve the alignment problem. This thesis also proposes architectures to implement the derived multi-character AC-DFA, AC-NFA, and hybrid FA. Moreover, a multi-stage architecture with configurable scheme is also proposed to provide flexibility in the number of characters inspected in parallel and in the changing of keyword sets. Evaluations are performed for the proposed architectures, respectively, to show their effectiveness. The main contributions of this work can be summarized as follows.

- A systematic approach that performs concatenation operations iteratively is devised to derive multi-character transitions from the transitions of the AC-algorithm, in which assistant transitions are introduced to solve the alignment problem.

- Several architectures are proposed to implement the derived multi-character transitions. The features of the AC-algorithm that provide a matching output in every matching cycle are preserved by introducing priority multiplexers to determine the corresponding matching output for all inspected characters in every matching cycle in the proposed multi-character string matching engines.

– The space required for implementing a derived k-character AC-NFA is $O(k)$. The proposed multi-character hybrid AC-FA approach has the feature that the number of transitions grow nearly linearly with respect to the number of characters under inspection in parallel.

– A configurable architecture is devised from the proposed multi-stage architecture that can process different keyword sets by simply updating the configuration data.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 introduces related string-matching literature. Chapter 3 first describes the AC-algorithm briefly and then presents approaches for implementing the AC-algorithm. Next, Chapter 4 proposes the derivation algorithm for multi-character FA and Chapter 5 describes the implementations of the derived multi-character FAs. Chapter 6 proposes configurable architectures for multi-character string matching. Chapter 7 compares the proposed approaches with related works. Conclusions are finally drawn in Chapter 8.

Chapter 2

Related Work

Among the string-matching algorithms, the algorithms of Aho-Corasick [1] and Bloom [8] are used in many applications for filtering out specific data efficiently. The Bloom algorithm accelerates the string matching by allowing for a small number of falsely matched patterns; however, a further exact verification is required for confirming whether the result is false positive. The Bloom algorithm can be implemented in hardware with high space efficiency [9]. In contrast, the AC-algorithm is an exact string-matching algorithm that can locate multiple patterns in a text with linear time complexity. Since the proposed approaches in this thesis are mainly based on the AC-algorithm, the following discussion of related work focuses on the AC-algorithm.

2.1 Optimization in Hardware

Due to the progress and flexibility of the programmable devices such as FPGA, developers can design and evaluate variant architectures according to the features of the AC-algorithm. Nevertheless, the resources of programmable devices are limited, with some works attempting to increase the hardware efficiency. To improve the memory efficiency, Tuck et al. [2] developed a bitmap-compression and path-compression approach for the AC-algorithm, capable of reducing the required memory and improving the performance on hardware implementation. Zha and Sahni [3] improved the bitmap-compression and path-compression approach by requiring considerably less memory. Alternatively, Alicherry et al. [4] implemented the AC-algorithm by integrating a Ternary Content Addressable Memory (TCAM) and a Static Random-Access Memory (SRAM) that utilizes ternary matching of TCAM to achieve the matching of characters expressed in negation expressions, subsequently reducing the space required for storing the state transitions. Lin and Liu [10] and Pao et al. [11] applied pipeline architectures to implement the character trie that only contains goto functions of the AC-trie to reduce the space introduced by the failure functions. Hua et al. [12] developed another approach based on a block-oriented scheme instead of the usual byte-oriented processing of patterns to minimize the memory usage. Dimopoulos et al. [13] developed a Split-AC algorithm that partitions a whole AC-trie into multiple smaller tries to increase memory efficiency.

Several researches developed hybrid string-matching approaches combining DFA and NFA to obtain both the advantages of DFA and NFA while avoid their disadvantages. The previous hybrid approaches include exact and regular-expression string matching approaches. For example, Yang et al. [14] proposed the head-body finite automaton (HBFA) that partitions an AC-trie to a head DFA (H-DFA) and a body NFA (B-NFA) to improve the performance of soft implementation of the AC-algorithm. Becchi et al. [15] proposed a hybrid-FA solution for regular-expression string matching to bring together the strengths of both DFA and NFA, i.e. space efficiency and a deterministic hardware-implementation. In the hybrid-FA of Becchi et al., the nodes causing state explosion retain an NFA encoding, while the rest are transformed into DFA nodes.

2.2 Multi-Character Hardware Approaches

Because of the flexibility of programmable devices, some works have developed string-matching architectures that can inspect multiple characters in parallel to multiply the throughput of string matching. However, developing an approach capable of inspecting multiple characters in parallel must consider both the complexity and the alignment problem incurred in k-character matching processes. As an extension of the AC-algorithm, Sugawara et al. [16] proposed a string-matching method called Suffix-Based Traversing (SBT) to process multiple input characters in parallel and reduce the lookup table size. Alicherry et al. [4] proposed a k-compressed AC-DFA to achieve a parallel k-character matching engine. A k-compressed AC-DFA consists only of the states whose depth is a multiple of k in the original AC-trie and the leaf states of the original AC-trie, where the alignment problem is solved by using additional shallow states. Other works used multiple FSMs to achieve parallelism and solve the alignment problem, where each FSM is responsible for processing a pattern beginning at a different position, respectively [5–7]. However, those approaches require specific logics to combine thematching results from the FSMs. The approaches of Yamagaki et al. [17] and Katashita et al. [18] solve the alignment problem by using additional states and transitions.

2.3 Software Approaches

Recently, many software implementations of the AC-algorithm utilize the power of the multicore in CPU or GPU to accelerate string matching. For example, Scarpazza et al. [19] proposed an optimized algorithm for the IBM Cell/B.E processor, which is a heterogeneous multicore processor comprised of a 64-bit processor core and eight synergistic processor cores, to achieve high-performance exact string matching. In that algorithm, keywords are split to fit in the local memories of the processing cores to reach extremely high throughput for each processor.

However, Salmela et al. [20] developed a software approach capable of processing multiple characters in parallel. That approach uses short substrings of length q, referred to as q-grams, which process q characters as a single character, and bit parallelism to increase filtering efficiency. Nevertheless, their approach is designed to match a set of keywords with the same length. Because of advanced semiconductor technologies, multiple processing cores can be packaged in a single CPU or GPU chip. Recently, many software implementations of the AC-algorithm use the power of the multicore in CPU or GPU to accelerate string matching. For example, Scarpazza et al. proposed an optimized algorithm for the IBM Cell/B.E processor, which is a heterogeneous multicore processor comprised of a 64-bit processor core and eight synergistic processor cores, to achieve high-performance exact string matching [19, 21]. In that algorithm, keywords are split to fit in the local memories of the processing cores to reach extremely high throughput for each processor. Yang et al. [14] and Yang and Prasanna [22] derived an approach using a head-body finite automaton (HBFA) to improve the match ratio on multicore processors and implements the HBFA in multiple threads on the multicore system to achieve high throughput. Villa et al. presented a software approach for the AC-algorithm on the Cray XMT multithreaded shared memory machine, capable of achieving a throughput of 28Gbps [23]. The approach of Tumeo et al. assigns different packets to different CUDA/GPU threads, as proposed by NVIDIA, to increase the efficiency of pattern matching [24]. Tumeo et al. later evaluated several software implementations of the AC-algorithm on shared and distributed memory architectures [25]. Herath et al. applied multicore CPUs to accelerate the string matching used in biology applications [26].

Software and hardware approaches significantly differ in achieving the parallelism. Software approaches achieve parallelism by splitting an input text into multiple chunks and then processing the chunks by multiple threads, respectively, where each thread still inspects the input text character by character. Conversely, hardware approaches achieve parallelism by inspecting multiple characters in parallel. Both approaches can multiply the throughput of string matching. However, software and hardware approaches also differ in that the former can have a larger dictionary size.

Chapter 3

Aho-Corasick Algorithm and Implementations

The algorithm proposed by Aho and Corasick (AC-algorithm) is an exact string-matching algorithm that can locate the occurances of multiple patterns in a text with a linear time complexity. Therefore, the AC-algorithm is used in many applications to fast filter out the specific data. This chapter first explains the AC-algorithm briefly and then presents various implementations of the AC-algorithm.

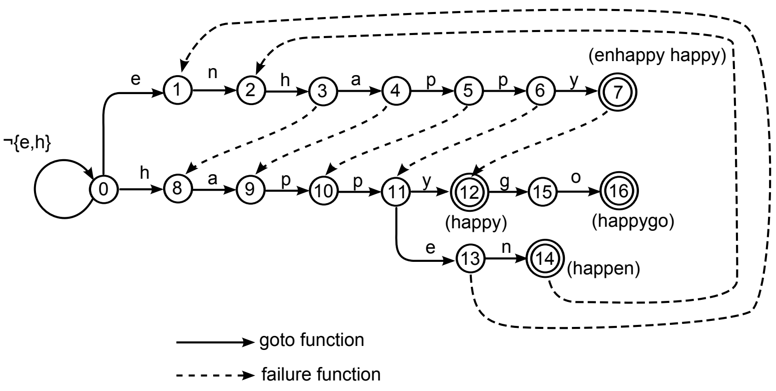


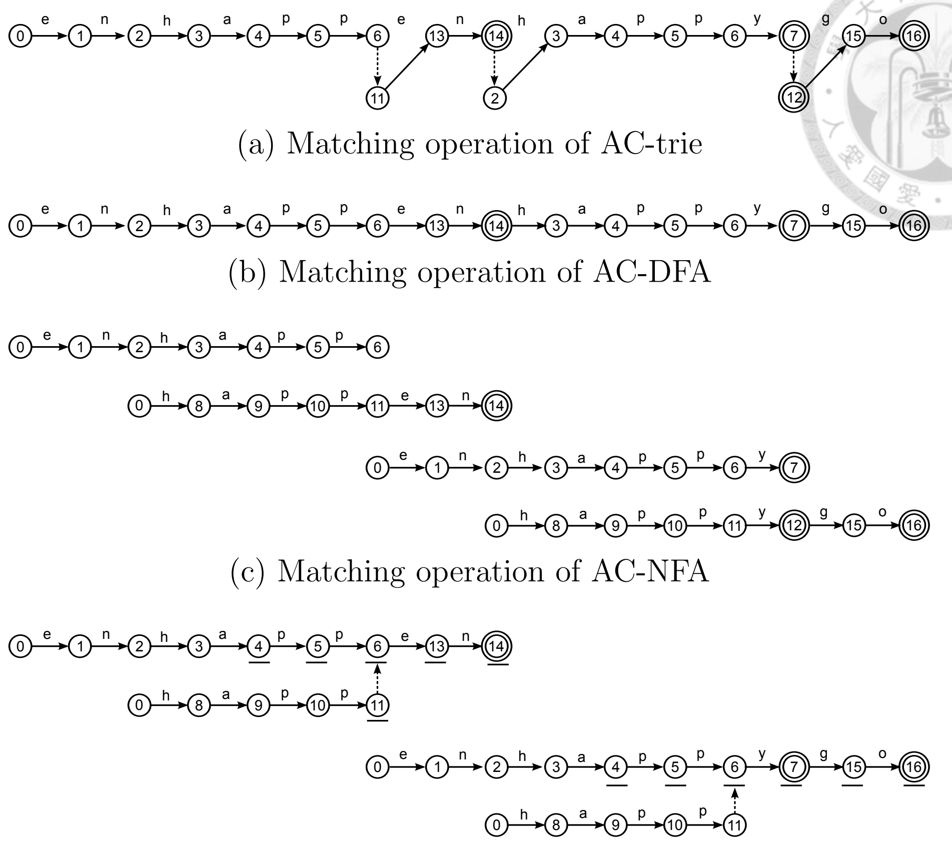
Figure 3.1: AC-trie

3.1 AC-trie

A prefix-tree built according to the AC-algorithm is known as an AC-trie that consists of goto and failure functions. Fig. 3.1 illustrates an AC-trie built on the keyword set {enhappy, happy, happen, happygo}. In this figure, the circled numbers denote states, which are called nodes alternatively. In addition, the double circled numbers denote output states, or output nodes, that have non-empty matching outputs. The physical and dashed lines denote goto and failure functions respectively. State 0 is also known as the initial state or the root node. Every non-initial state has a failure function, while the failure functions linked to the initial state are not shown for clarity. In an AC-trie, the depth of a node is also called the level of it. A property of the failure function is that a state only links to another state in the smaller level through the failure function. For example, the failure function of state 7 in level 7 links to state 12 in level 5.

A matching cycle in the AC-algorithm is defined as a period that begins with inputting a character and ends with outputting a matching output. There is only one active state in an AC-trie at any time. In a matching cycle, the goto functions of the active state are checked first. If none of the goto functions is matched, then the state transits to a new state through the failure function and the goto functions of the new activated state are checked continuously. Since all non-initial states are linked to the initial state eventually through the failure functions and the initial state has the goto functions for all characters, it ensures that a matched goto function can be found in every matching cycle. In the matching process, an input string is processed character by character and we call the character under processing as an inspecting character. A matching cycle leads to a matching output, which is represented by a state number. If a keyword is matched, the matching output is a non-zero state number; otherwise the matching output is state 0.

The AC-algorithm can be implemented in various approaches, which are described later, and the matching operations of these approaches are illustrated together in Fig. 3.2 for comparison. The inspecting texts of these matching examples are the same which is ‘enhappenhappygo’. The matching operation of the AC-trie is illustrated in Fig. 3.2(a). In response to the characters ‘enhapp’, the transition traverses through the states 0, 1, 2, 3, 4, and 5 sequentially. In response to the next character ‘e’, none of the goto functions of state 5 matches with this character, so that the state transits to 11 following the failure function of state 5 and the operation continues to match the goto functions of state 11. In response to the following character ‘n’, the state transits to 14 which is a output state and we have a matching output ‘happen’. State 14 is a terminal state, therefore according to its failure function, the state transits to 2 when the next character is input and then the matching process goes on from the state 2. Similarly, in response the remaining characters, the transition traverses through the states 3, 4, 5, 6, 7, 12, 15, and 16 sequentially;



(a) Matching operation of AC-trie

(b) Matching operation of AC-DFA

(c) Matching operation of AC-NFA

(d) Matching operation of hybrid AC-FA

Figure 3.2: Examples of matching operations

where the transition from 7 to 12 is a transition according to the failure function of state 7. In the remaining matching operation, we have matching outputs on states 7 and 16 which are ‘enhappy happy’ and ‘happygo’ respectively.

With the failure functions, all matched keywords can be found in a one-pass search by using an AC-trie. Nevertheless, when an AC-trie is implemented in hardware the complexity will be increased due to the property that often more than one state transition are needed to find a matched goto function through the failure functions. An AC-trie can be implemented as a DFA (AC-DFA) or an NFA (AC-NFA). The DFA and NFA approaches have significant different features. This thesis proposes a hybrid finite automaton approach that combines the DFA and NFA approach to implement the AC-algorithm (hybrid AC-FA). The proposed hybrid AC-FA approach has both the advantages of DFA and NFA approaches. The AC-DFA, AC-NFA, and hybrid AC-FA approaches are described in the following sections.

3.2 AC-DFA

Fig. 3.3 illustrates the DFA converted from the original AC-trie. The transitions pointed to the states 1 and 8 that are derived from the failure functions are denoted together, respectively, for clarity in this figure. Fig. 3.2(b) illustrates the matching operations of the AC-DFA which is straightforward and the explanation of the matching operations is omitted. Fig. 3.4 illustrates the block diagram for implementing the AC-DFA. The transition table is implemented as a lookup table generally. The next state NX is determined by the input character IN_CHAR, current state CUR_ST, and the transition table. The next state NX is saved in a register and output as as the matching result OP. The next state NX is looped back as the current state CUR_ST to be used in the next matching cycle. However, the transition table of AC-DFA typically is a sparse table, and thus it is inefficient in space utilization.

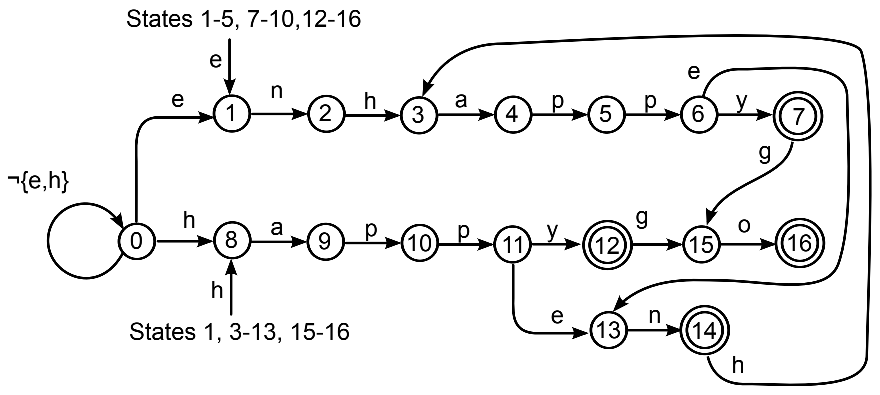


Figure 3.3: AC-DFA

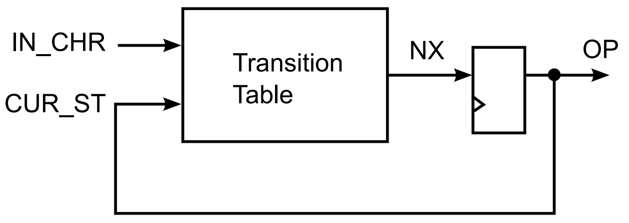


Figure 3.4: Basic AC-DFA implementation block diagram

Alicherry et al. has analyzed the AC-DFA constructed for the signatures obtained from 100 viruses in the Internet [4]. Table 3.1 summarizes the number of states at a certain depth and the number of transitions to the states at that depth in the AC-DFA. The analysis result indicates that the number of transitions to the states at depth 1 is as high as 96.4% of the total transitions. Notably, most of the transitions to smaller depths are mainly the next transitions derived from failure functions.

Table 3.1: States and transitions in an AC-DFA (Alicherry et al.)

Depth (d)	1	2	3	4	5	≥ 6	Total
States	56	89	100	102	102	7248	7698
Transitions	421,243	7,576	551	165	121	7382	437,038

Accordingly, Alicherry et al. proposed an alternative AC-DFA implementation that uses a TCAM and a SRAM to implement the transition table, in which a wildcard character is used to resolve the problem of transition explosion. Each TCAM entry maps a transition in the AC-DFA to an index; and the TCAM index is used to compute the address of the corresponding memory block in the SRAM. In a matching cycle, the TCAM-based approach first obtains a matching index from the TCAM according to the inspecting character and current state, and then obtains a next state from the SRAM according to this matching index.

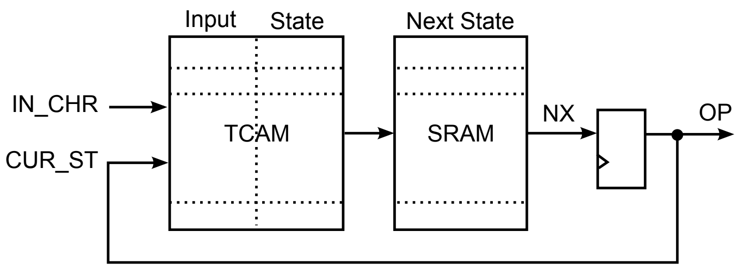


Figure 3.5: Implementation of AC-DFA with TCAM and SRAM

Since a TCAM only outputs indexes according to the matching results, an additional SRAM is required for obtaining next states from the resulting indexes in the TCAM-based approach. This thesis develops a hardware architecture, shown in Fig. 3.6, that can achieve the feature as integrating the TCAM and SRAM to implement the AC-DFA. This proposed architecture consists of multiple rule units and these rule units can be configured according to the keyword set to be processed, in which each rule unit processes a transition respectively.

Two transitions beginning from the initial state are $\delta_1(0,e)=1$ and $\delta_1(0,h)=8$. In order to be distinguishable with a multi-character transition which will be described

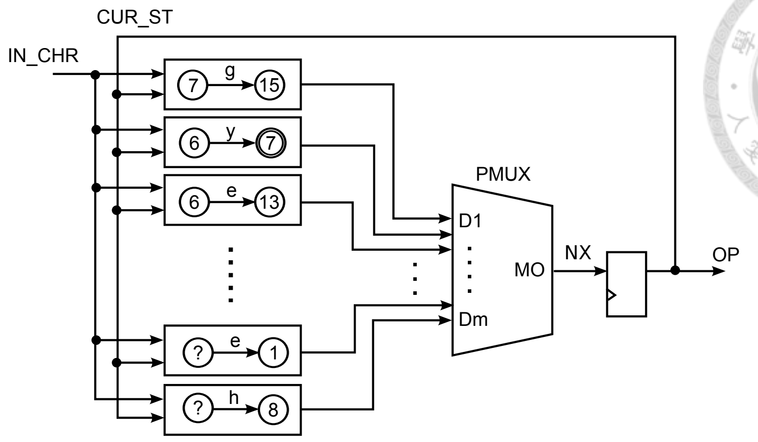


Figure 3.6: Implementation of AC-DFA with priority multiplexer

later, the notation δ_1 with a subscript 1 means a 1-character transition. In this figure, the initial state is replaced by a wildcard character '?', and thus the two transitions of the initial state are replaced by δ_1(?,e)=1 and δ_1(?,h)=8. In this way, the transitions linked to states 1 and 8 derived from the failure functions can be omitted, such as δ_1(1,e)=1, δ_1(1,h)=8, and so on. Because of the wildcard character, multiple rules may be activated simultaneously. For example, when the current state is 6 and the input character is 'e', the transitions δ_1(6,e)=13 and δ_1(?,e)=1 are activated simultaneously.

In a typical TCAM, the conflict situation caused by multiple potential results is resolved by using a priority encoder to determine a final matching index. In this work, the conflict situation arising from multiple activated rules is resolved by using a priority multiplexer to determine a final matching result. In Fig. 3.6, the priority multiplexer PMUX has m inputs D1 through Dm. If the inputs Di and Dj are both valid and $i < j$, then the priority of Di is higher than that of Dj; and thus, the priority multiplexer selects the input Di to be output through MO. For PMUX, input D1 has the highest priority, and input Dm has the lowest priority. When none of the inputs is valid, PMUX outputs a default value, such as 0; this ensures that the next state should return to the initial state when none of rules is matched. In Fig. 3.6, the upper rule unit has a higher priority. As a result, when the conflict situation that δ_1(6,e)=13 and δ_1(?,e)=1 are activated simultaneously is happen, the priority multiplexer PMUX determines the next state is 13.

3.3 AC-NFA

If the failure links are removed and simultaneously activation of multiple states is allowed, an AC-trie becomes an NFA. Fig. 3.7 illustrates the AC-NFA obtained from the original AC-trie by removing the failure links. After converting an AC-trie to an AC-NFA, all matched transitions are done currently. The parallelism implicit in hardware makes it more feasible to keep track of concurrent state transitions. The transitions of an AC-NFA are only the goto functions of the original AC-trie. In the proposed approach, the complexity in terms of number of transitions remains the same, whereas the failure functions are transformed into the concurrent transitions that fit the hardware intrinsically.

Fig. 3.2(c) illustrates the matching example of the AC-NFA with the same input 'enhappenhappygo'. In response to the characters 'enhapp', a transition sequence begins from state 0 and ends at state 6, which does not match any keyword. In response the characters beginning from the third character, another transition sequence traverses states 0, 8, 9, 10, 11, 13, and 14, which matches the keyword 'happen'. The characters beginning from the seventh character initialize a transition sequence traverses states 0 to 7 and matches the keyword 'enhappy'. Similarly, the characters beginning from the ninth character initialize a transition sequence traverses states 0, 8, 9, 10, 11, 12, 15, and 16, which matches the keyword 'happygo'. As can be seen by comparing Fig. 3.2(a) and (c), in every matching cycle, when a state is activated in the AC-trie, all the states linked to through the failure functions from the active state are activated simultaneously in the AC-NFA. For example, when state 3 is activated, state 8, which is pointed by the failure function of state 3, is activated simultaneously. Therefore the failure functions are not necessary if multiple states are allowed to be activated simultaneously.

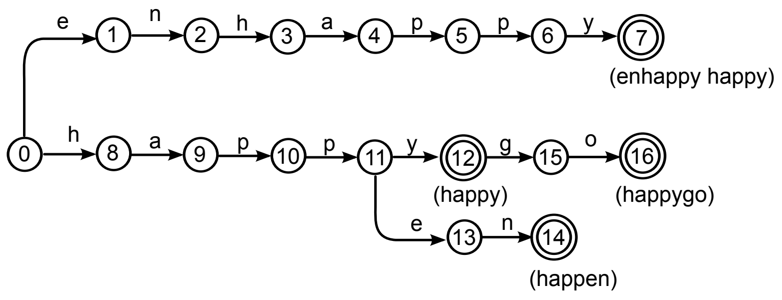


Figure 3.7: AC-NFA

Fig. 3.8 illustrates the implementation of the AC-NFA, where some similar por-

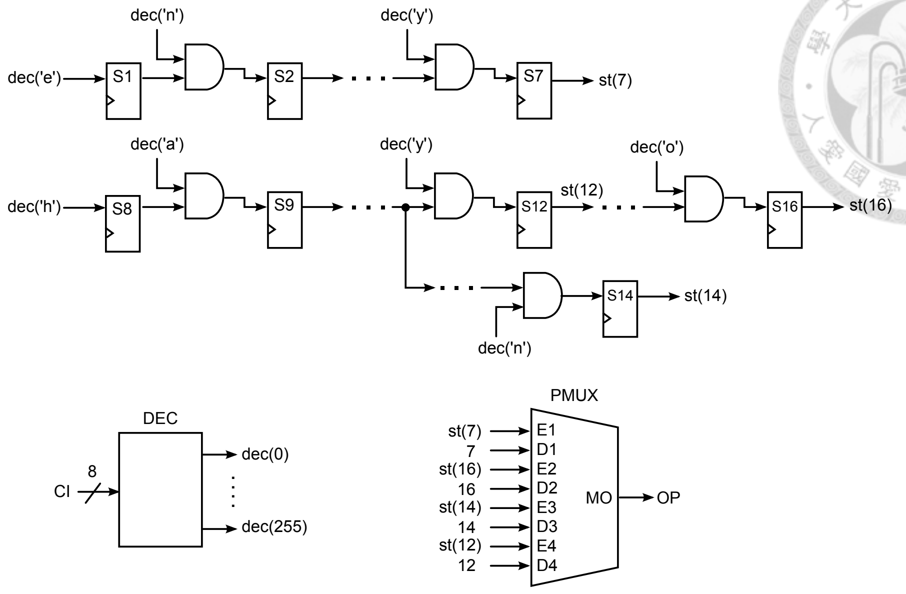


Figure 3.8: Implementation of AC-NFA

tions are omitted in the circuit for clarity. The AC-algorithm provides only one matching output in every matching cycle, while multiple states may be activated simultaneously in an AC-NFA. Therefore, an output circuit is required to determine a matching output in the implementation of an AC-NFA. In this implementation, the final matching output OP is determined by using a priority multiplexer PMUX. Notation st(i) denotes the status of node i, where i represents the state number in the AC-NFA. For example, when the text under inspection is 'enhappy', states 7 and 12 are activated simultaneously. State 7 represents the string 'enhappy' and state 12 represents the string 'happy'; the former includes the later. Therefore, state 7 has a higher priority and is determined as the matching output.

In an AC-trie, each state represents a unique string. If a failure function links state S_1 to state S_2, then the string represented by S_2 is the postfix of the string represented by S_1. For example, state 12 represents the string 'happy', and state 7 represents the string 'enhappy'; in addition, the failure function of state 7 points to state 12, and the string 'happy' is the postfix of the string 'enhappy'. In an AC-trie, the matching output is simply the active state since only one state is activated at any time. Although activation of multiple states is allowed in an AC-NFA, the proper matching output from the multiple active states must be determined. For example, like the earlier case, states 7 and 12 are activated simultaneously. Since failure functions link higher states to lower-level states, the highest-level activated state in an AC-NFA should determine the final matching output.

In the matching operation of the AC-algorithm, only a matching output is generated after every matching cycle. In the proposed NFA approach, the priority multiplexer PMUX shown in the lower right portion of Fig. 3.8 is used as an output selection circuit to determine the final matching outputs from multiple activated output nodes. Notably, the priority multiplexer PMUX in this figure differs from that in Fig. 3.6 in that each input group consists a control signal and a data signal; nevertheless, they are the same essentially and the difference in diagrams is convenient for explanation only.

This AC-NFA has four output nodes so that the priority multiplexer PMUX has four input groups (E1, D1) through (E4, D4), where inputs E1 through E4 are control signals and inputs D1 through D4 are data signals. The control signals E1 through E4 indicate whether the data inputs D1 through D4 are valid or not, respectively. If the inputs Ei and Ej are both true and $i < j$, then the priority of input Ei is higher than that of input Ej; in addition, the priority multiplexer selects the input Di to be output through MO. When none of the inputs is valid, the output MO is not valid either. However, the output MO of PMUX can output 0 if no matched output is available. Notation st(s) refers to the status of node s, which is true when node s is activated. Since higher-level nodes have a higher priority, signal st(9) has the highest priority and is connected to E1. The data sent to inputs D1 through D4 are the corresponding state numbers. Consider the previous example. Following acceptance of the text 'enhappy', both nodes 7 and 12 are activated and both st(7) and st(12) become true. Moreover, since the priority of st(7) is higher than that of st(12), PMUX selects the data 7 input from D1 as the matching output.

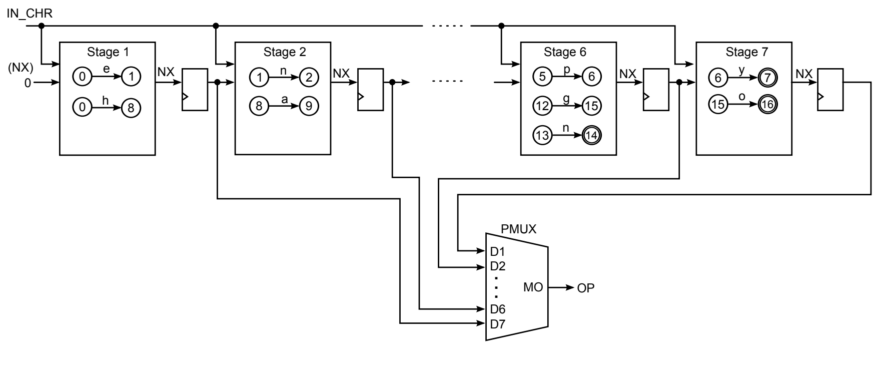


Figure 3.9: Multi-stage architecture of AC-NFA

According to the observation, among the states with the same depth in an NFA derived from a given AC-trie, the number of active states n_a is no more than one, i.e. $n_a \leq 1$. Therefore, an AC-NFA can be implemented in a multi-stage architecture alternatively. In an AC-trie, the states with the same depth are known as in the same level. The states with the same depth represent different strings with the same length, and thus at most one state in a level can be activated in one time. Consequently, only one register is required to save the active state for each level in an AC-NFA. Fig. 3.9 illustrates a multi-stage architecture for implementing the AC-NFA. The multiple stages are arranged in a chain, in which each stage includes the transitions of the corresponding level. Fig. 3.10 illustrate the block diagram of a stage unit. A stage unit includes multiple rule units, in which each rule unit is responsible for matching one transition. Therefore, the number of rule units in a stage must be equal or greater than the number of transitions. A rule unit contains the information of its corresponding transition and matches the information with the input current state CUR_ST and character IN_CHAR in the matching operation. A rule unit is triggered when its pattern is matched with the inputs and then outputs the next state NX according to the transition.

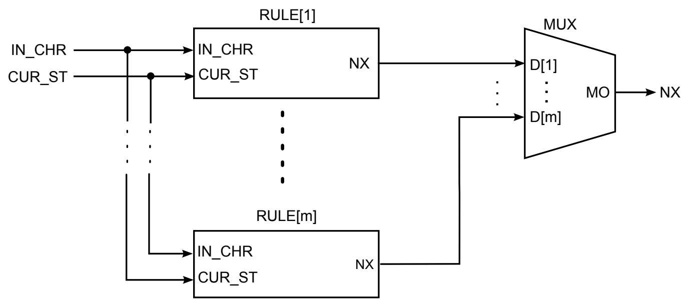


Figure 3.10: Block diagram of a stage unit

3.4 Hybrid AC-FA

The DFA approach has an attractive property that processing an input string involves one DFA state traversal per character, which implies a deterministic number of memory accesses. Namely, the memory bandwidth requirement for implementing a DFA is predictable and it is possible to implement a DFA in a lookup table approach. However, the number of transitions grows explosively when an AC-trie is converted to a DFA. In contrast, an NFA approach is efficient in the hardware size utilization. However, an NFA is difficult to be implemented in a pre-designed architecture, like a lookup table approach, since each input character can trigger multiple state transitions and multiple states can be active simultaneously. Generally, an NFA is suitable to be implemented in a programmable device.

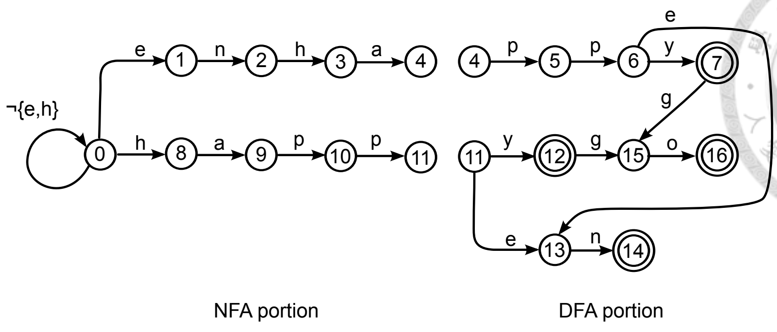


Figure 3.11: Hybrid AC-FA

According to the analysis result in Table 3.1, the transitions increase dramatically in the lower levels due to the failure functions when an AC-trie is converted to a DFA. Accordingly, this work proposes a hybrid finite automaton to implement an AC-trie (Hybrid AC-FA), which has both the advantages of DFA and NFA. Fig. 3.11 shows a hybrid AC-FA that is obtained by dividing the AC-trie in Fig. 3.1 to NFA and DFA portions. In the NFA-portion, all failure functions are removed and only goto functions are remained; while in the DFA-portion, the failure functions are substituted by expanding goto functions. For the convenience of discussion, let the NFA levels are defined as the number of levels in NFA portion, in which level 0 is excluded. For example, there are three NFA levels in the hybrid AC-FA illustrated in Fig. 3.11. Notably, states 4 and 11 are in the DFA portion.

Comparing with the AC-DFA illustrated in Fig. 3.3, the transitions linked backwardly to states 1, 3, and 8 are eliminated in the hybrid AC-FA. In which, most of the removed transitions are linked backwardly to the states in levels 0 and 1, i.e. states 0, 1 and 8. Correspondingly, the hybrid AC-FA only has two more transitions, linked to states 13 and 15, respectively, than the AC-NFA illustrated in Fig. 3.7.

Like the multi-stage architecture for implementing the AC-NFA, only one register is required for saving the state for each level in the NFA portion. Furthermore, because at most one state is allowed to be activated in the DFA portion, only one register is required for keeping the state in the DFA portion. As a result, the stages and the registers required for keeping the states in the implementation of hybrid AC-FA can be predetermined, and this enables to design a general string-matching architecture based on the AC-algorithm. Because the transitions increase little as comparing with the AC-NFA approach, the hybrid AC-FA approach is efficient in space utilization.

Fig. 3.12 shows the multi-stage architecture for implementing the hybrid AC-FA.

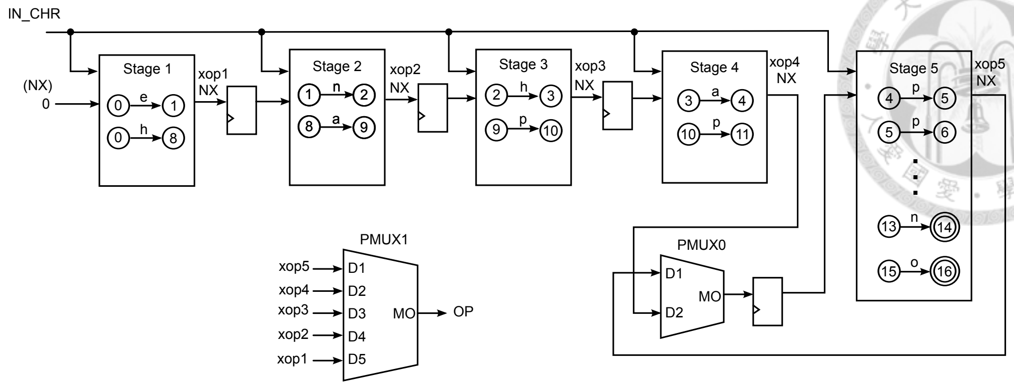


Figure 3.12: Implementation of hybrid AC-FA

Stages 1 to 4, which belong to the NFA portion, contain the transitions of levels 0 to 3 respectively. Stage 5, which is the terminal stage and belongs to the DFA portion, contains all of the transitions. Because only one state can be activated at most in each stage, priority multiplexer PMUX0 determines the next state for the terminal stage from the next states (NX) output from stage 4 and stage 5. The detail of each stage is same as the block diagram shown in Fig. 3.10. The next state (NX) generated by the *i*-th stage also represents the matching result xop*i* of that stage. PMUX1 determines the final matching output from the matching results xop1 to xop5.

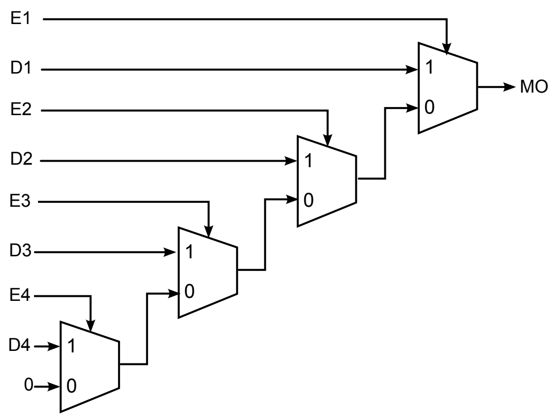


Figure 3.13: Priority multiplexer implemented by chained multiplexers

3.5 Priority Multiplexer

The priority multiplexer plays an important role in the proposed architectures and its implementation is described briefly here. The implementation of priority multiplexer mainly refers to the literature of Alera [27]. In a priority multiplexer, the select logic implies a priority, so the options to select the correct item must be checked in order. Fig. 3.13 illustrates a 4-to-1 priority multiplexer implemented by multiple chained multiplexers that evaluates each condition, or select bit, one at a time. However, this structure of chained multiplexers is generally bad for delay, since the critical path through the logic traverses through every multiplexer in the chain. As a result, the delay of the structure of chained multiplexers increases linearly respect with the number of data inputs.

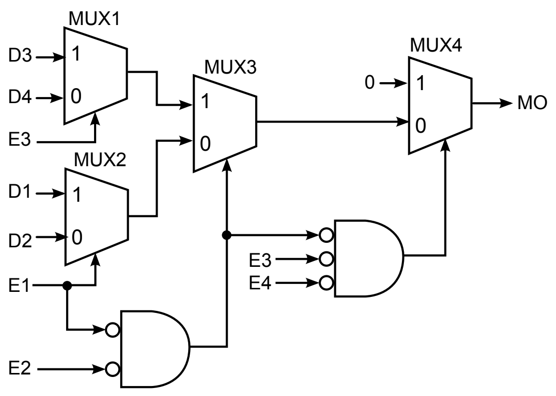


Figure 3.14: Priority multiplexer optimized for delay

Fig. 3.14 illustrates an alternate implementation of priority multiplexers to optimize the delay. This logic structure is just slightly more complicated than the standard priority multiplexer scheme, but significantly improves the delay through the logic. In this structure, if any of the two select lines E1 and E2 are high, then the 2-input AND gate chooses the upper half of the logic, otherwise it chooses the lower side. The enable signals E1 through E4 make the final choice of inputs, if all of the enable signals are low then the output MO is zero. The signal E1 has the highest priority in the figure. The levels of the chained multiplexers is $log_2 m$ for m data inputs in the optimized structure. Therefore the delay of the optimized priority multiplexer reduces from $O(m)$ to $O(log_2 m)$ for m data inputs.

Chapter 4

Multi-Character State Transitions

The throughput of a string-matching engine can be multiplied in the same operating speed when multiple characters can be processed in parallel in every matching cycle. Therefore, a 1-character finite automaton (FA), which processes only one character at a time, has to be derived to a multi-character FA for processing multiple characters in parallel. Furthermore, the alignment problem also has to be considered in the derivation of multi-character FA.

This chapter proposes an derivation algorithm to extend the AC-algorithm to be able to process multiple characters in parallel. This proposed algorithm can derive a 1-character FA, which could be an AC-DFA, an AC-NFA, or a hybrid AC-FA, to a multi-character FA by using iterative concatenation operations. In addition, assistant transitions and a virtual state are introduced to deal with the alignment problem in the derivation algorithm.

```
 Input: thishers
Match1: -his----
Match2: ---she--
Match3: ----he--
Match4: ----hers
```

Figure 4.1: Example of alignment problem

4.1 Alignment Problem and Previous Work

The alignment problem must be considered when multiple characters are inspected in parallel in every matching cycle. Fig. 4.1 uses an example to demonstrate the alignment problem. In which, the keyword set to be matched is {he, she, his, hers}.

In this matching example, eight characters are inspected in parallel and the input text is 'thishers'. As can be seen, all the four keywords appear in the inspecting text; however, all the occurrences of keywords do not begin at the first character of the inspecting text.

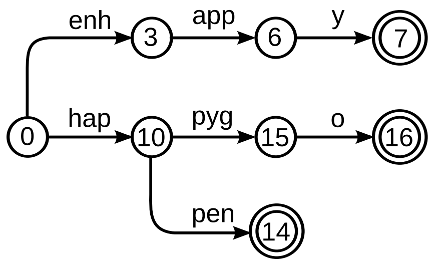


Figure 4.2: Example of 3-compressed FA

An intuitive approach to convert a 1-character FA to a multi-character FA is merging multiple successive 1-character transitions to a multi-character transitions, like the k-compressed FA proposed by Alicherry et al. [4]. An example of 3-compressed FA is shown in Fig. 4.2. However, because some states are lost in a k-compressed FA as comparing with the original AC-trie, special process is required to deal with the alignment problem. In the Alicherry approach, shallow nodes are added to resolve the alignment problem as shown in Fig. 4.3. When a keyword does not appear in the first character of the inspecting character-chunk, an additional transition is needed to ensure the next state to fall into an existing state in the compressed FA. For example, when the input text is 'penhapp', the state first transits from the initial state to the the shallow state 2 according to 'pen' and then to state 3 according to the additional character 'h'. Consequently, two transitions are required when alignment problem occurs. Moreover, multiple transitions may be activated simultaneously because of the same prefix or suffix, such as the keywords 'hers' and 'she'; this further increases the complexity in hardware design.

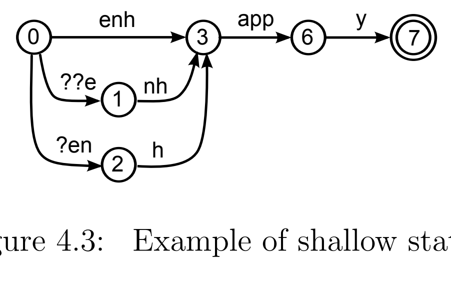


Figure 4.3: Example of shallow states

Alternatively, Tripp proposed a architecture consisting of multiple identical FSMs, as shown in Fig.4.4, to solve the alignment problem [5]. In which, each FSM is responsible to deal with a alignment case respectively. However, the multi-FSM architecture requires a combine logic circuit to determine the final matching results from the results of FSMs.

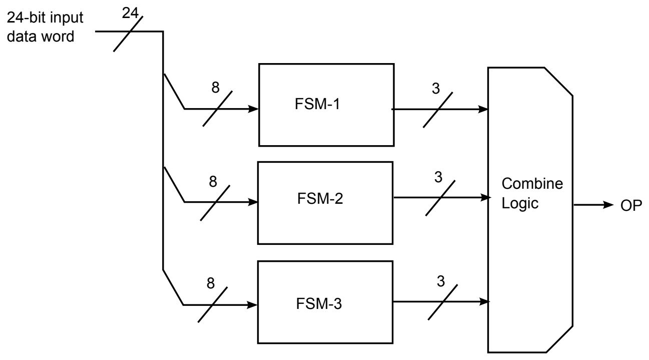


Figure 4.4: Architecture of three FSMs

4.2 Derivation of Multi-Character Transitions

This section proposes a derivation algorithm that derives 1-character transitions to multi-character transitions by using iterative concatenation operations. Before describing the concatenation operation, some related definitions are given as follows.

Definition 1. A k-character transition $\delta_k(S_1, T) = S_2$ represents the state transit from the current state S_1 to the next state S_2 on a k-character string T.

For example, a 3-character transition δ_3(10,pyg)=15 represents that the state transits from 10 to 15 on a 3-character string ‘pyg’.

Definition 2. A transition $\delta_k(S_2, T_2) = S_3$ is a successive transition of the transition $\delta_l(S_1, T_1) = S_e$ if $S_2 = S_e$, implying that the starting state of $\delta_k(S_2, T_2) = S_3$ is simply the end state of $\delta_k(S_1, T_1) = S_e$.

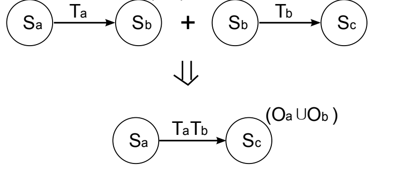


Figure 4.5: Concatenation operation

A new multi-character transition can be obtained by concatenating a transition with its successive transition. The concatenation operation is defined as follows and its diagram is shown in Fig.4.5:

Definition 3. (Concatenation of two transitions) Given a k-character transition $\delta_k(S_1, T_1) = S_2$ and an l-character successive transition $\delta_l(S_2, T_2) = S_3$, where S_1, S_2, and S_3 denote states, T_1 represents a k-character string, and T_2 represents an l-character string. Then the concatenation of the two transitions is a $(k+l)$-character transition $\delta_{k+l}(S_1, T_1T_2) = S_3$.

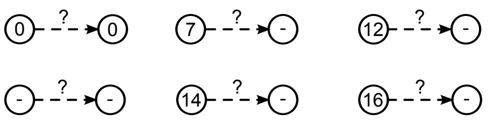


Figure 4.6: Assistant transitions

Besides the transitions of the 1-character FA, three types of assistant transitions are defined to facilitate the construction of multi-character transitions capable of solving the alignment problem.

Definition 4. (Assistant transitions) The first type of assistant transitions is defined as $\delta_1(0, ?) = 0$ that denotes a transition from state 0 to state 0 on an arbitrary character. The second type of assistant transitions is defined as $\delta_1(S_{op}, ?) = -$ that denotes a transition from an output state S_{op} to a pseudo state on an arbitrary character. The third type of assistants is defined as $\delta_1(-, ?) = -$ that denotes a transition from a pseudo state to another pseudo state on an arbitrary character.

Fig. 4.6 shows examples of assistant transitions. Assistant transitions are denoted by dashed lines to distinguish them from the 1-character transitions of a original FA. The circled character '-' denotes a pseudo state that is a virtually defined state and nonexistent in an AC-trie. Furthermore, the symbol '?' denotes an arbitrary character. The first type assistant transition, i.e. $\delta_1(0, ?) = 0$, deals with the alignment problem in which the beginning of a pattern does not appear in the first character of the inspecting characters. The second type assistant transition, i.e. $\delta_1(S_{op}, ?) = -$, preserves the matching output for a situation in which the ending of a pattern does not appear in the final character of the inspecting characters. In Fig.4.6, the transitions beginning from states 7, 12, 14, and 16, respectively, and ending at pseudo states are examples of the second type assistant transitions. The

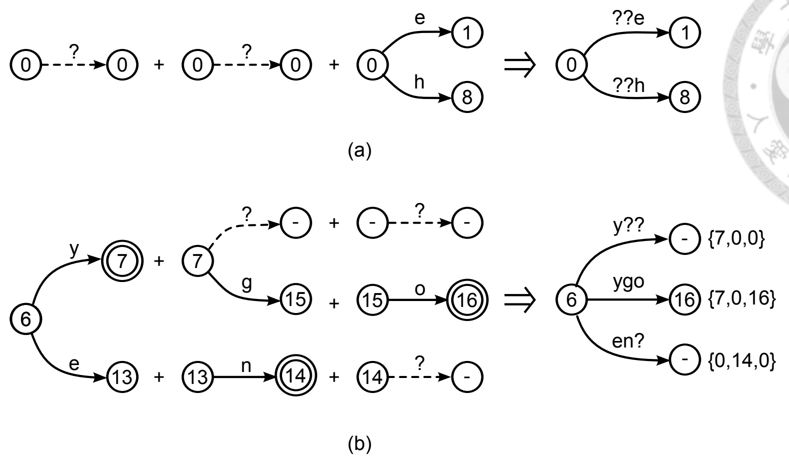


Figure 4.7: Derivation examples

third type assistant transition, i.e. $\delta_1(-,?) = -$, can follow a second type assistant transition to form a multi-character transition.

The examples shown in Fig. 4.7 explain how to derive 3-character transitions by repeating concatenation operations. These examples also explain the usages of the assistant transitions in the derivation. In this figure, '+' denotes the concatenation operation.

First, referring to Fig. 4.7(a), concatenating two assistant transitions δ_1(0,?)=0 obtains a 2-character transition δ_2(0,??)=0; and then concatenating the deriving transition with δ_1(0,e)=1 and δ_1(0,h)=8, respectively, obtains two 3-character transitions δ_3(0,??e)=1 and δ_3(0,??h)=8. Despite obtaining a 3-character transition δ_3(0,???)=0, concatenation of the transition δ_2(0,??)=0 with another δ_1(0,?)=0, it is discarded owing to its uselessness. The transitions δ_3(0,??e)=1 and δ_3(0,??h)=8 imply that the state stays at state 0 for the first two any characters and then transits to state 1 or 8 on the third character 'e' or 'h', respectively. By using these derived transitions, the alignment problem arising from the situation that the first character of keywords appears at the third inspecting character is dealt with.

In the example of Fig. 4.7(b), state 6 has two goto functions, δ_1(6,y)=7 and δ_1(6,e)=13. Concatenating δ_1(6,y)=7 with two assistant transitions δ_1(-,?)=- obtains a 3-character transition δ_3(6,y??)=-, and concatenating δ_1(6,y)=7 with other two transitions δ_1(7,g)=15 and δ_1(15,o)=16 obtains another 3-character transition δ_3(6,ygo)=16. Concatenating δ_1(6,e)=13 with δ_1(13,n)=14 and δ_1(-,?)=- obtains a 3-character transition δ_3(6,en?)=-. States 7 and 14 are output states, particularly state 14 is a terminal state, each of which is concatenated with assistant transitions to form a complete 3-character transition to preserve the matching output. Because the trailing two characters of the pattern in transition δ_3(6,y??)=- are wildcard characters, the transition is always triggered along with the transition δ_3(6,ygo)=16; this conflict situation is resolved by introducing a priority multiplexer in the proposed hardware implementation. As a result, repeating the described concatenating operation iteratively can derive all k-character transitions from a given AC-trie for any required number k.

From an AC-NFA, we can infer that when the characters under simultaneous inspection are increased by one more, the number of transitions increased is equal to the number of output states. This observation is owing to that each output state is concatenated with an assistant transition to preserve the matching output for a situation that the following character is not matched. Next, consider the transition δ_1(6,y)=7 as an example, concatenating it with the successive transition δ_1(7,g)=15 can obtain δ_2(6,yg)=15 and concatenating it with the assistant transition δ_1(7,?)=- can obtain δ_2(6,y?)=-; the latter transition δ_2(6,y?)=- is used to preserve the matching output for the situation that only the first character is matched with 'y'.

An NFA consisting of the k-character transitions derived from an AC-NFA is called as a k-character AC-NFA here. The proposed approach can be used to derive a k-character AC-NFA for any number k from an original AC-NFA, which is called 1-character AC-NFA alternatively. The growth of the derived k-character transitions is linear with respect to the number k. As a result, the number of the k-character transitions grows linearly with respect to k, the number of the characters to be inspected in parallel.

The result can be summarized as the following description. For a given AC-NFA with the number of 1-character transitions denoted as r_1 and the number of output states denoted as n_{op}, then the number of the k-character transitions is obtained as follows:

$$r_k = r_1 + (k-1) \times n_{op} \tag{4.1}$$

For instance, the AC-NFA example has 16 1-character transitions and four output states, and thus the number of 3-character transitions is $16 + (3-1) \times 4 = 24$.

Algorithm 1: Algorithm for deriving k-character transitions

```
   Input  : $k$: number of characters
            $NXSET$: 1-character transition set
   Output: $TRSET$: $k$-character transition set
 1 begin
 2     $TRSET \leftarrow$ empty
 3     for each state $Si$ do
 4     begin
 5         $NSET \leftarrow$ all 1-character transitions of $Si$ in $NXSET$
 6         repeat $k-1$ do
 7         begin
 8             $TMPSET \leftarrow$ empty
 9             for each transition $NXi$ in $NSET$ do
10             begin
11                 $NX\_ST \leftarrow$ next state of $NXi$
12                 for each transition $NXj$ of $NX\_ST$ do
13                 begin
14                     $NEW\_TR \leftarrow$ concatenate $NXi$ with $NXj$
15                     $TMPSET \leftarrow TMPSET \cup NEW\_TR$
16                 end
17             end
18             $NSET \leftarrow TMPSET$
19         end
20         $TRSET \leftarrow TRSET \cup NSET$
21     end
22     remove unused transitions from $TRSET$
23     return $TRSET$
24 end
```

4.3 Derivation Algorithm

Algorithm 1 is a generalized algorithm for deriving multi-character transitions from an AC-trie. In this algorithm, the input parameter k is the number of characters to be inspected in parallel. The input parameter $NXSET$ contains the original 1-character transitions that are derived according to the approach of AC-DFA, AC-NFA, or hybrid AC-FA. The output variable $TRSET$ contains the resulting k-character transitions. By using multiple level iterations, this algorithm derives the k-character transitions for every state in the original AC-trie.

In line 2, $TRSET$ is initialized. Statements in the loop between line 3 and line 21 derive all of the k-character transitions of every state Si. In line 5, the 1-character transitions of state Si are duplicated to variable $NSET$. The loop between line 7 and line 19 is repeated $k-1$ times, in which the 1-character transitions of Si are concatenated with their successive 1-character transitions iteratively to derive the k-character transitions of Si. After executing the loop, $NSET$ contains all of the k-character transitions of Si. In line 20, $NSET$ is added to $TRSET$. The algorithm then returns to line 5 to process the next state continuously. The algorithm is terminated when all of the states are processed. Finally, $TRSET$ contains the derived k-character transitions.

Now let's look into the loop between line 7 and line 19. In line 8, $TMPSET$ is initialized. In the loop between line 10 and line 17, every transition NXi in $NSET$ is expanded. In line 11, the next state of NXi is assigned to NX_ST. In the loop between line 13 and line 16, NXi is concatenated with every transition NXj respectively beginning from NX_ST to obtain new transitions. In line 14, NXi is concatenated with NXj to obtain a new transition NEW_TR. In line 15, NEW_TR is added to $TMPSET$. Here the number of the pattern characters of NEW_TR is one more then the number of the pattern characters of NXi.

Because the intermediate state is concealed after two transitions are concatenated, the matching outputs must be reserved in the concatenation operation in line 14. Moreover, some transitions consists of all assistant transitions that may be obtained in the concatenation process are not useful and are subsequently removed in line 22.

Now let's estimate the time complexity of the derivation algorithm by using k-character AC-NFA. According to equation (4.1), the number of transitions in a i-character AC-NFA is $r_i = r_1 + (i-1) \times n_{op}$, implying that r_i concatenation operations are required to derive the i-character transitions from (i-1)-character transitions. Therefore, the total required concatenation operations T_c to derive the 1-character transitions to k-character transitions can be obtained as follows:

$$T_c = \sum_{i=2}^{k}(r_1 + (i-1) \times n_{op}) = (k-1) \times r_1 + \frac{2k}{k-1} \times n_{op}$$

Consequently, the time complexity of Algorithm 1 is $O(k \times r_1 + n_{op})$. Since a derived multi-character AC-DFA or hybrid AC-FA generally has more transitions than a multi-character AC-NFA for the same original AC-trie, the time complexity for deriving a multi-character AC-DFA or hybrid AC-FA should be greater than that for deriving a multi-character AC-NFA.

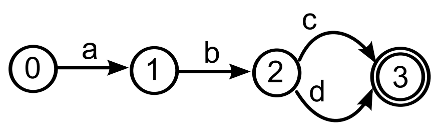


Figure 4.8: Regular-expression example

4.4 Comparison of Derivation Approaches

Some previous researches have proposed derivation approaches for deriving multi-character transitions in different perspectives. This section discusses the approaches of Yamagaki et al. [17] and Yang et al. [28] for comparison. Since both their approaches derive the transitions of a regular expression to multi-character transitions, the regular expression "`ab(c|d)`" is used as an example for comparison; the FSM built on the regular expression example is shown in Fig. 4.8.

4.4.1 The proposed approach

First, the FSM example is derived to a 2-character FSM by using the proposed approach and the derivation process is briefly shown in Fig. 4.9. At the beginning, the 1-character transitions of the FSM are retrieved and the assistant transitions are prepared as shown in Fig. 4.9(a). In which, the transitions denoted as dashed lines are assistant transitions. Every transition is concatenated with its successive transitions to obtain 2-character transitions; the concatenation processes are shown Fig. 4.9(b). Finally, the resulting 2-character FSM is shown in Fig. 4.9(c).

4.4.2 Yamagaki et al. approach

Yamagaki et al. derive multi-character transition by iteratively merging two successive transitions and solve the alignment problem by adding additional transitions for the initial state and all final states. Fig. 4.10 describes a derivation example of Yamagaki et al. [17]. In this example, the 1-character NFA of "`ab(c|d)`" is converted into a 2-character NFA. Fig. 4.10(a) shows the NFA graph obtained from the regular-expression example, where dashed edges show original edges. Before the derivation procedure, a self edge $\delta_1(0,?)$=0 is added to the initial node and an additional transition $\delta_1(S_f,?)$=- is added for each final node S_f. The derivation procedure merges each of the transitions with its successive transition(s) respectively. These tasks are performed for all of the transitions, and then the 2-character NFA graph as shown in Fig. 4.10(b) is obtained. Fig. 4.10(c) shows the rearranged FSM to show how the

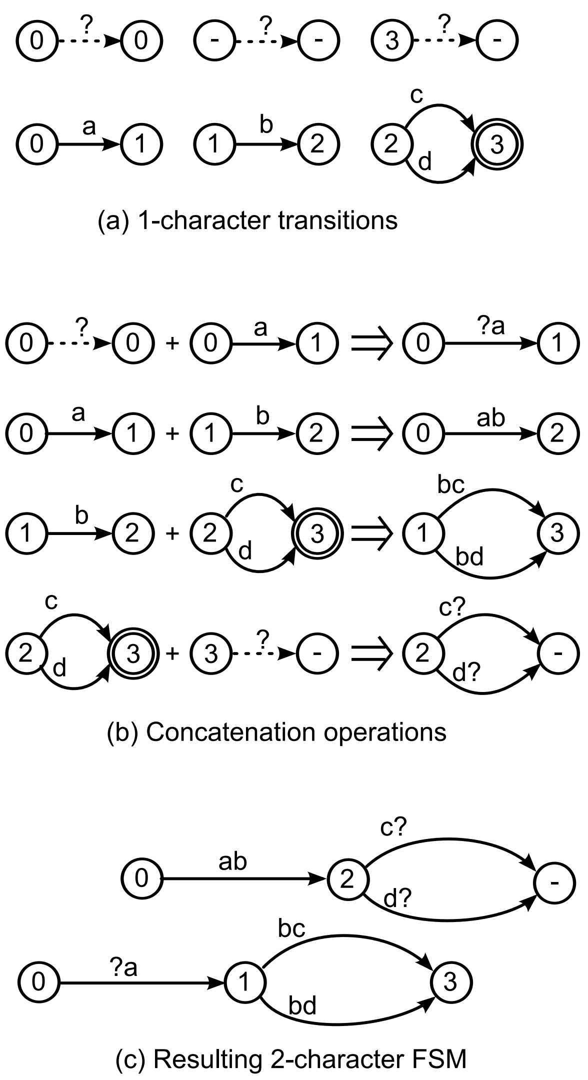

(a) 1-character transitions
(b) Concatenation operations
(c) Resulting 2-character FSM
?
a
b
c
d
?a
ab
bc
bd
c?
d?

Figure 4.9: Deriving 2-characters NFA by using the proposed approach

alignment problem is resolved. Where the right final state, i.e. the added virtual node, becomes active when a match occurs at the first position of the two input character positions, and the left state, i.e. state 3, becomes active when a match occurs at the second character position. If matches occur at both positions, both states become active. Similarly, a 4-character NFA can be obtained from the new derived 2-character NFA by the merging operations. Consequently, a 2^k-character NFA can be obtained by k iterations of the described derivation procedure.

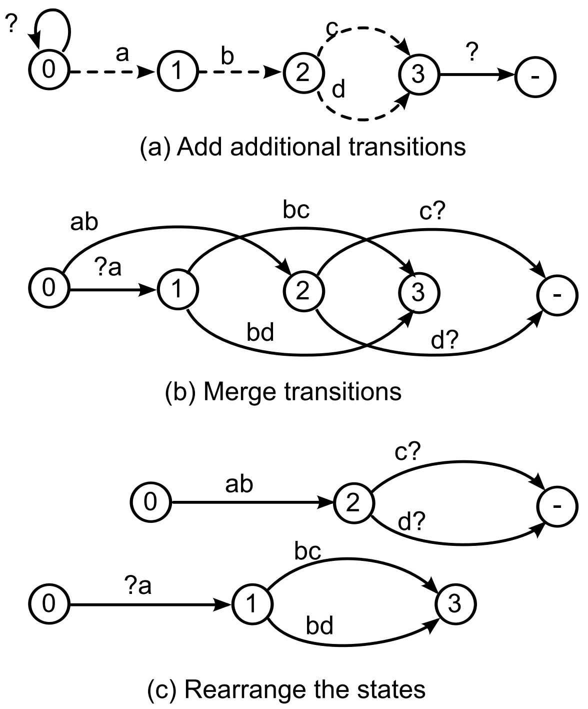


Figure 4.10: Deriving 2-character NFA by using Yamagaki et al. approach

4.4.3 Yang et al. approach

Yang et al. proposed a circuit-level spatial approach to construct a multi-character regular-expression matching engine [28]. Their approach accepts an m-character matching-engine circuit C_m and a p-character matching-engine circuit C_p for the same regular expression, and then merges the two input circuits by reconnecting the transitions to obtain an $(m+p)$-character matching-engine circuit C_{m+p} for the same regular expression. In the merging procedure, the states of one of the input circuits are converted to dummy nodes, and then the destination of each of the links is reconnected to the same destination node in the other circuit. Notably, the two input-circuits and the output circuit have the same states. As a result, merging two identical 1-character FSMs obtains a 2-character FSM; and merging the resulting 2-character FSM and the original 1-character FSM further obtains a 3-character FSM. Repeating the procedures can obtain an FSM for processing a required number of characters in parallel.

An example of the merging procedure proposed by Yang et al. is shown in Fig. 4.11. In (a), FSM1 and FSM2 are two identical original FSMs. The nodes and transitions of FSM2 are denoted as grey color for clarity. In addition, the states in FSM2 are converted to dummy nodes. Fig. 4.11(b) illustrates the FSM after

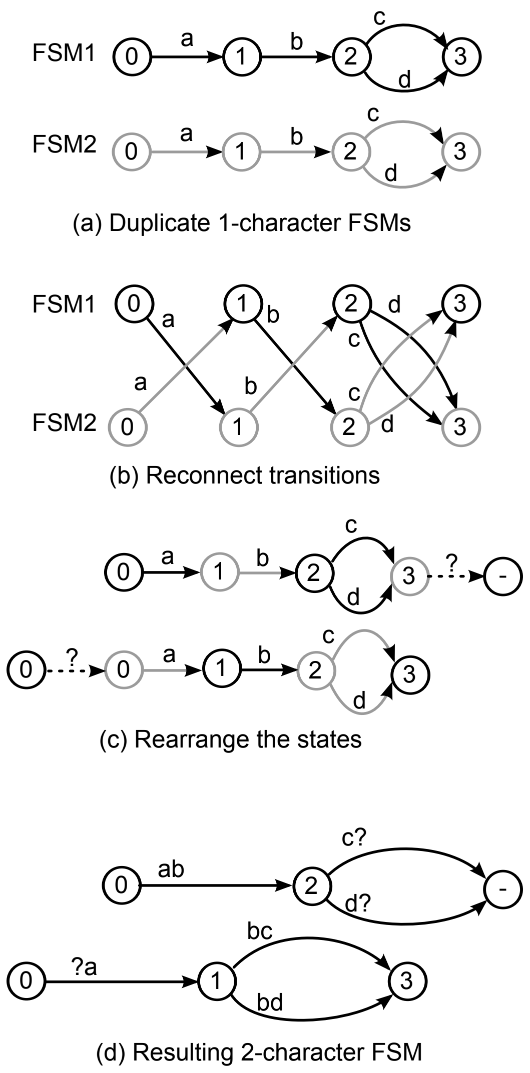

FSM1
FSM2
(a) Duplicate 1-character FSMs
FSM1
FSM2
(b) Reconnect transitions
(c) Rearrange the states
ab
c?
d?
?a
bc
bd
(d) Resulting 2-character FSM

Figure 4.11: Deriving 2-character NFA by using Yang et al. approach

reconnecting the destination nodes. For example, the destination node of $\delta(0, a) = 1$ in FSM1 is reconnected to state 1 in FSM2; and, on the other hand, the destination node of $\delta(0, a) = 1$ in FSM2 is reconnected to state 1 in FSM1. The resulting FSMs after rearranged are shown in Fig. 4.11(c), where additional transitions are added for obtaining complete 2-character transitions. Finally, Fig. 4.11(d) shows the FSMs after removing the dummy nodes.

4.4.4 Summary of comparison

As can be seen in Fig. 4.10, 4.11, and 4.9, the resulting 2-character FSMs are all the same. While the derivation approach of Yamagaki et al. achieving by merging operations does have the limitation that the number of characters to be inspected in parallel is restricted to 2^k. In addition, the wildcard character is used to deal with the alignment problem in these approaches. Although the paper of Yang et al. does not mention how to resolve the alignment problem, the wildcard character is used implicitly in resolving the alignment problem in the circuit derived by using their approach.

Furthermore, the obtained 2-character FSM has the same states as the original FSM has, excepting the pseudo states added for dealing with the alignment problem. Moreover, the number of transitions increases linearly with respect to the number of transitions ended with final states, because of the assistant transitions added for preserving the matching information.

Chapter 5

Multi-Character String-Matching Approaches

This chapter proposes various architectures for implementing the multi-character FAs, including AC-DFA, AC-NFA, and hybrid AC-FA, derived by the algorithm described in previous chapter. In addition, the proposed approaches are implemented in FPGA devices to evaluate the utilization of hardware resource and estimate achievable throughput.

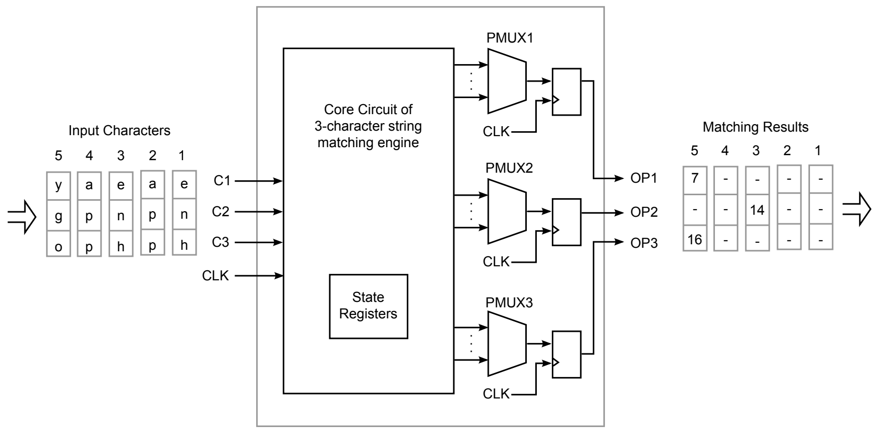


Figure 5.1: Basic block diagram of a 3-character string matching engine

First, the generalized block diagram and matching operation of a multi-character string matching engine is described. This generalized block diagram and operation are applicable to the proposed architectures described in this thesis. Fig. 5.1 shows the complete bock diagram of a basic 3-character string matching engine, and Fig. 5.2 shows the waveform of the matching example with an input text ‘enhappenhappygo’.

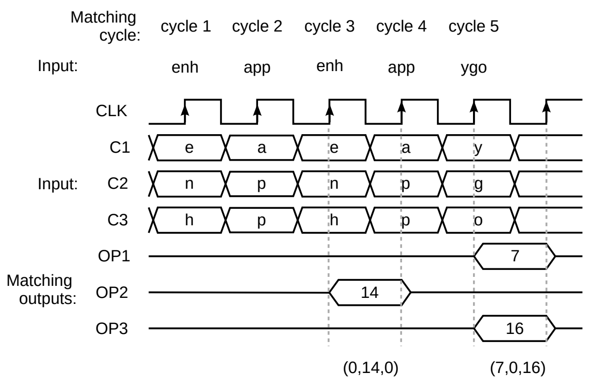


Figure 5.2: Waveform of matching operations

In every matching cycle, three characters are input via C1, C2, and C3 in parallel; and then, three corresponding matching outputs are generated from OP1, OP2, and OP3 after this matching cycle. In most of the implementations in this thesis, a matching cycle is equal to a clock cycle. Three registers are connected after PMUX1, PMUX2, and PMUX3 to save the matching outputs. A matching output can include a signal to indicate it is valid or not; the matching output is plotted as a zero line if it is invalid in the waveform.

The matching operations are synchronized with the clock signal CLK and data are latched in the rising edges of CLK. This input text is divided to five 3-character chunks ‘enh’, ‘app’, ‘enh’, ‘app’, and ‘ygo’ and then processed in five consecutive matching cycles, respectively. An inspecting 3-character chunk is input via IN_CHRS, which includes C1 through C3, and three corresponding matching outputs are generated from OP1 through OP3 after one matching cycle. A valid matching output OP2, which is 14 or ‘happen’, after matching cycle 3; and two valid matching outputs OP1 and OP3, which are 7 and 16 or ‘happy’ and ‘happygo’ respectively, after matching cycle 5.

5.1 Multi-Character AC-DFA

This section proposes an efficient architecture for implementing the derived multi-character AC-DFA. Fig. 5.3 illustrates the transitions of 3-character AC-DFA. In which, the growth of transitions is reduced by using a wildcard character to represent a unmatched character. Because the intermediate states are concealed when multiple characters are inspected simultaneously, the output set is denoted beside the pattern in a transition. For example, the output set denoted beside δ_3(5,py?)=- is (0,17,-

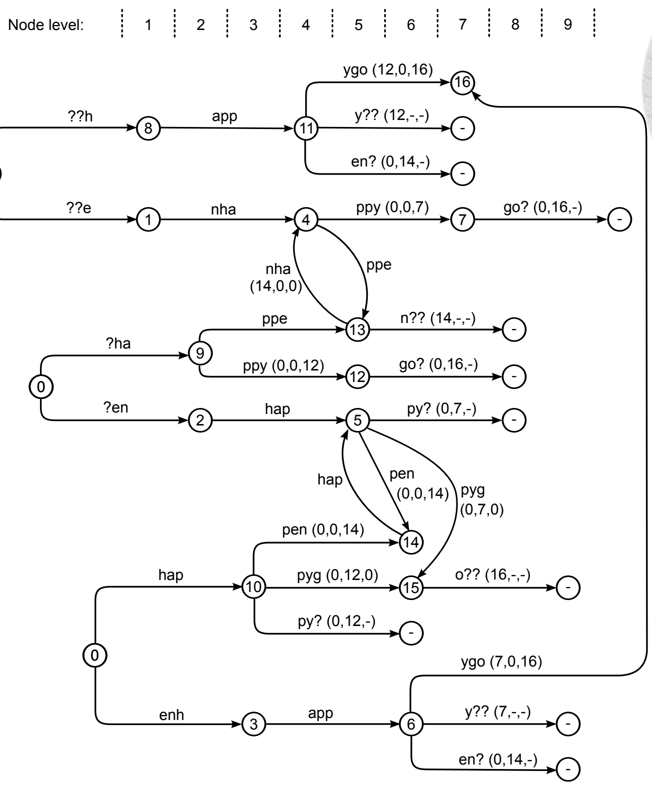


Figure 5.3: Transitions of the 3-character AC-DFA

), in which the three notations of this output set are corresponding to the three pattern characters respectively. The first number 0 represents a empty string; and the second number 17 means state 17 which represents keywords 'enhappy happy'. Finally, the last notation '-' represents a invalid output which is corresponding to the wildcard character '?' in the pattern. Notably, an output set is not denoted if all of the matching outputs are empty strings for clarity.

The 3-character AC-DFA is depicted as three portions according to the levels of nodes. The first to the third groups, from top to bottom, consist of the nodes in the $(3 \times i + 1)$, $(3 \times i + 2)$, and $(3 \times i)$ levels, in which i is an integer and $i \geq 0$. Some transitions across two groups are derived from failure functions.

5.1.1 Implementation

Fig. 5.4 illustrates the block diagram of the implementation of 3-character AC-DFA. The string matching engine accepts the input IN_CHRS, which consists of three characters C1 through C3, and produces three matching outputs OP1 through OP3 corresponded to the three input characters, respectively. In the application of the

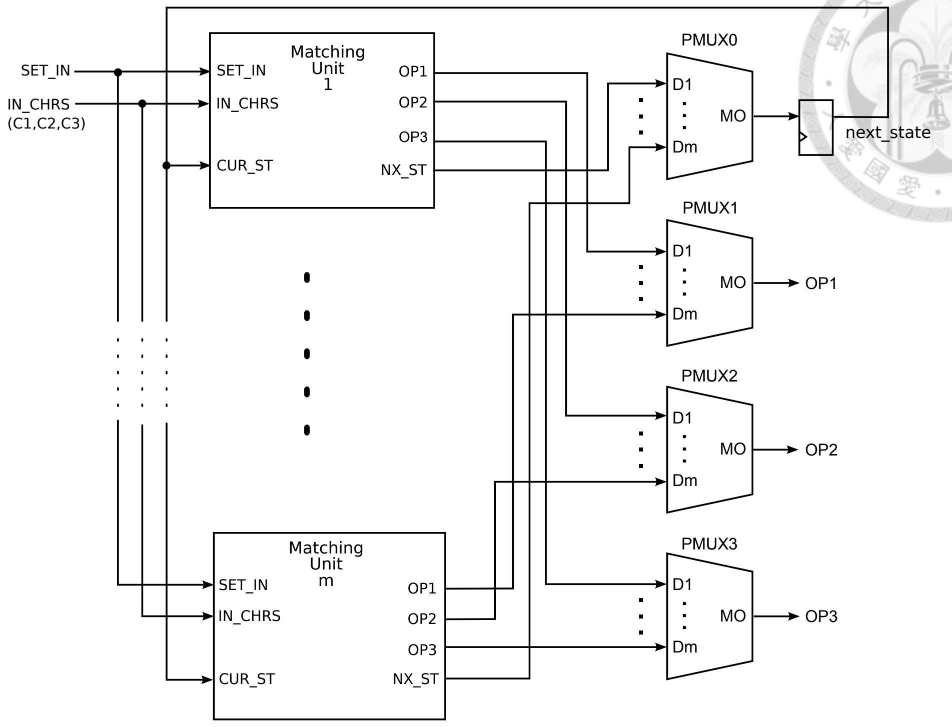


Figure 5.4: Architecture for a 3-character AC-DFA

proposed string-matching engine, an inspecting text is split to chunks of three characters and then fed via the input IN_CHRS to this device chunk by chunk. This string matching engine includes m matching units where each matching unit is responsible for processing a transition. Because the derived 3-character transitions include wildcard characters and multiple transitions could be activated simultaneously, priority multiplexers are needed to determine the next state and matching outputs. The priority multiplexer PMUX0 determines the next state and PMUX1 through PMUX3 determine three final matching outputs. When none of the inputs of a priority multiplexer is valid, this priority multiplexer output a default output, such as zero; and thus it ensures that a next state can be return to the initial state when none of rules is matched.

The detailed diagram of each matching unit is illustrated in Fig. 5.5. Where each matching unit includes data registers, a matching circuit, and a control circuit. The block enclosed by dashed lines is a control circuit. The control circuit which consists of multiple AND gates performs AND operation of the output EQ of the matching circuit with the corresponding bits of output mask OMASK to obtain the control flags NX_FLG and OF1 to OF3. The information of transition rule is stored in the data registers. The data registers are logically partitioned to two groups which store

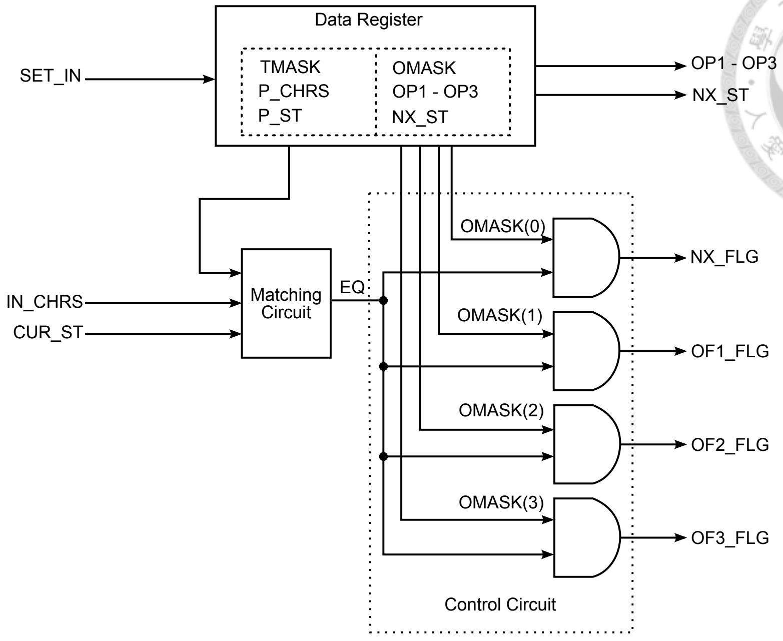


Figure 5.5: Block diagram of a matching unit

the pattern data and the output data, respectively. The matching circuit matches the input data with the pattern data. If the matching result is matched, then the output EQ is true; otherwise the output EQ is false.

The pattern data include the ternary mask TMASK, the pattern characters P_CHRS, and the current state P_ST. The output data include the output mask OMASK, the matching outputs OP1 to OP3, and the next state NX_ST. The details of pattern data and output data are explained together with the description of the transition rules.

Table 5.1 lists the transition rules of the example 3-character AC-DFA. The example 3-character AC-DFA has total 32 3-character transition rules while only portion of the rules are listed as examples. In the table, '-' represents a don't-care output, while in the real work the matching output is determined by the corresponding bit in OMASK. The rule number in the first field is only for the sake of convenience in explanation and is not required to be stored in registers in the real implementation. In the table, the rules are arranged in the order according to their priorities; where rules 1 through 32 are arranged from the highest priority to the lowest priority.

The fields of each rule can be grouped to pattern data and output data roughly. The pattern data include ternary mask TMASK, current state P_ST, and pattern characters P_CHRS. The output data include output mask OMASK, next state

Table 5.1: Transition rules of the 3-character AC-DFA

Rule	Patern Data			Output Data				
no.	TMASK	P_ST	P_CHRS	OMASK	NX_ST	OP1	OP2	OP3
1	1100	15	o??	0100	-	16	-	-
2	1111	14	hap	1111	5			
3	1111	13	nha	1111	4	14		
4	1100	13	n??	0100	-	14	-	-
5	1110	12	go?	0110	-		16	
				:				
				:				
28	0111	0	hap	1111	10			
29	0011	0	?en	1011	2	-		
30	0011	0	?ha	1011	9	-		
31	0001	0	??e	1001	1	-	-	
32	0001	0	??h	1001	8	-	-	

NX_ST, and matching outputs OP1 through OP3. The pattern data are compared with the input characters and the current state, if the comparing result is matched then the rule is activated and the output data with flag are generated. The matching outputs OP1 through OP3 are corresponded to the first to third characters of the pattern characters P_CHRS respectively. Each bit of the ternary mask TMASK determines if the corresponding pattern data should be compared or not. For example, the bits of TMASK, from the most significant bit (MSB) to the least significant bit (LSB), are corresponding to the current state P_ST, and the first to third pattern characters of P_CHRS, respectively.

A pattern character is compared when the corresponding ternary mask bit is '1'; otherwise the pattern character is don't-care. For example, the bit 3 of TMASK in rules 28 through 32 are all '0', it means the current states P_ST of these rules are don't-care. Each bit of the output mask OMASK determines if the corresponding output data is valid or not. For example, the bits of OMASK, from the MSB to the LSB, in the rules are corresponding to the next state NX_ST, and the matching outputs OP1 to OP3 respectively. For example, bit 3, bit 1, and bit 0 of the OMASK in the first rule are all '0', which means the NX_ST, OP2, and OP3 are not valid; namely, when rule 1 is activated it does not affect both the next state and the matching outputs corresponded to the second and third input characters.

In Table 5.1, the matching outputs are expressed as state numbers; this facilitates a hardware implementation. For example, the output OP1 of the first rule is 16 which is corresponded to the output string ‘happygo’, and the output OP1 of the fourth rule is 14 which is corresponded to the output string ‘happen’. In this manner, the outputs can be stored in fixed width spaces instead of storing the variable length string data, and it is convenient for hardware design.

Now, the hardware cost required for implementing a k-character AC-DFA by using the proposed architecture is estimated. Suppose that each input character is b_c bits, P_ST and NX_ST are b_s bits, and each of OPi is b_p bits in the implementation. In addition, the widths of TMASK and OMASK are $k+1$ bits. As a result, the total width of k matching outputs is $k \times b_p$ bits and the width of P_CHRS is $k \times b_c$ bits. The required space b_m of each matching unit for storing one k-character transition rule is calculated as follows:

$$\begin{aligned} b_m &= (k+1) + b_s + k \times b_c + (k+1) + b_c + k \times b_p \\ &= (b_c + b_p + 2) \times k + 2b_s + 2 \end{aligned} \tag{5.1}$$

As can be seen from the above equation, the space for storing the rule in each matching unit is proportional to the number of characters, i.e. k, to be inspected in parallel. If the usual 8-bit character set is used and the matching outputs are represented by state numbers, then $b_c = 8$ and $b_p = b_s$, and the required space can be obtained by the following simplified equation:

$$b_m = (b_s + 10) \times k + 2b_s + 2 \tag{5.2}$$

As can be seen from this equation, the number of states determines the widths of the registers and further determines the space for rules and complexity of comparators.

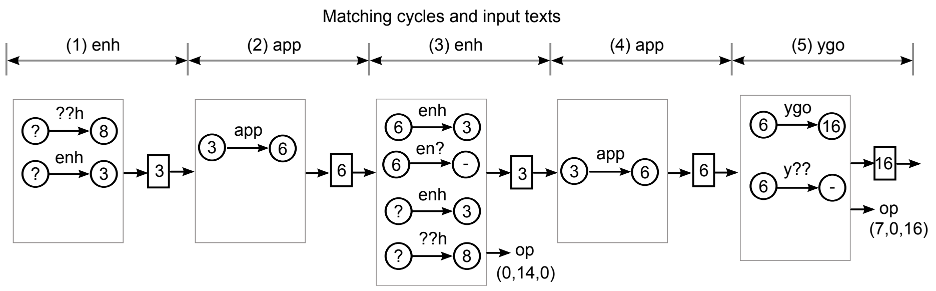


Figure 5.6: Matching example of the 3-character AC-DFA

5.1.2 Matching example

In order to demonstrate the effectiveness of the proposed multi-character AC-DFA approach, Fig. 5.6 illustrates an example of the matching operation of the 3-character AC-DFA. In this example, the input string to be matched is 'enhappenhappygo', which is divided to five 3-character chunks 'enh', 'app', 'enh', 'app', and 'ygo' and then processed in five consecutive matching cycles, respectively. This figure only displays the triggered transitions; the transitions with higher priorities are arranged in the upper portion. The matching results of a matching cycle are denoted under the next state, which are represented by state numbers for clarity. However, the matching results are omitted if all of them are empty results for clarity.

The state register is initialized before the matching procedure. In the first matching cycle, according to the input characters 'enh', the matched transitions are δ_3(0,enh)=3 and δ_3(0,??h)=8; in which, δ_3(0,enh)=3 has a higher priority and determines the next state to be 3. Both the matching outputs determined by these two triggered transitions are empty strings. In the second matching cycle, according to the input characters 'app' and the next state determined in the previous cycle, the matched transition is only δ_3(3,app)=6. The matching output determined by this triggered transition is an empty string in the second matching cycle.

In the third matching cycle, according to the input characters 'enh' and the next state, i.e. 6, determined in the previous matching cycle, the matched transitions are δ_3(6,en?)=-, δ_3(0,enh)=3, and δ_3(0,??h)=8. The matching result of δ_3(6,en?)=- is (0,14,-), which has a matching output corresponding to the second inspecting character. Consequently, at the end of the third matching cycle, the final matching result OP2 is 14, or 'happen', and both the other two results OP1 and OP3 are empty strings. The next state is determined as 3 in the third cycle. In the same way, OP1 through OP3 are all empty strings and the next state is 6 at the end of the fourth matching cycle. In the fifth matching cycle, two transitions δ_3(6,ygo)=16 and δ_3(6,y??)=- are triggered; the former has a higher priority and determines the matching results OP2 and OP3 are 7 and 16 respectively. The next state is determined as 16 in the fifth cycle.

5.1.3 Experiment and evaluation

The keywords retrieved from the rules of SNORT are used for evaluation. The numbers of transitions for k-character AC-DFAs of 200, 400, 600, 800, and 1,000 keywords are evaluated for the cases of k= 1, 2, 3, and 4.

Table 5.2: Rules for different k-character AC-DFA

keywords (total length)	$k=1$ (r1/len)	$k=2$ (r2/r1)	$k=3$ (r3/r2)	$k=4$ (r4/r3)
200 (2,321)	4,775 (2.06)	12,053 (2.52)	31,030 (2.57)	75,712 (2.44)
400 (4,692)	16,011 (3.41)	61,653 (3.85)	222,899 (3.62)	777,428 (3.49)
600 (8,032)	35,064 (4.37)	169,307 (4.83)	757,859 (4.48)	3,306,210 (4.36)
800 (10,728)	53,756 (5.01)	308,207 (5.73)	1,571,908 (5.10)	7,806,812 (4.97)
1000 (13,374)	73,465 (5.49)	458,623 (6.24)	2,590,140 (5.65)	14,249,657 (5.50)

len: total lenth of keywords
$r1$ to $r4$: numbers of rules for $k = 1$ to 4

Table 5.2 lists the numbers of transition rules for different number of keywords and different k-character AC-DFAs. The numbers in the parentheses, for $k = 1$ is the ratio of the rules for $k = 1$ to the total length of keywords, and for $k = 2$ through 4 are the ratios of the rules between k and $k - 1$. In this table, *len* represents the total length of keywords, and r_1 through r_4 represent the numbers of transition rules of k-character AC-DFAs for $k = 1$ through 4.

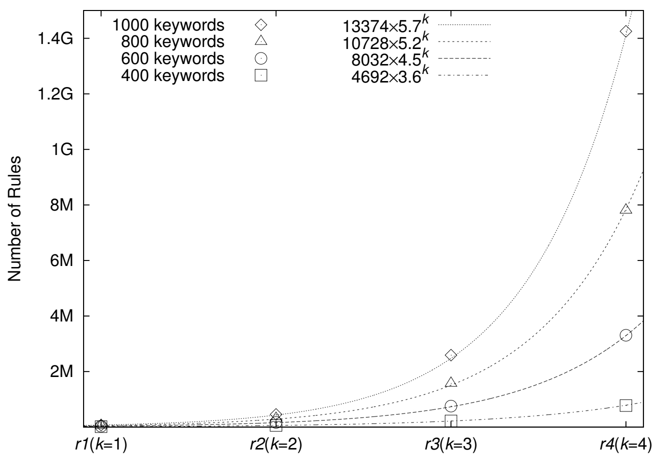


Figure 5.7: Growth of transitions of k-character AC-DFAs

Fig. 5.7 shows the growth of transitions of k-character AC-DFAs with respect k. These curves are represented by $r_k = len \times p^k$, where the value of p is dependent on the keyword set. The values of p are 2.4, 3.6, 4.5, 5.2, or 5.7 for 200, 400, 600, 800, and 1,000 keywords respectively. The value of p is obtained by the arithmetic mean of the ratios r_1/len and r_i/r_{i-1}, in which i is 1 through 4. The curve for 200 keywords is omitted in this figure since it is too close to the x-axis as compared with others. As can be seen in the results, the number of rules increases more rapidly when the keywords increase. This phenomenon might be owing to the fact that when keyword-set size grows the failure functions linked to non-initial states increase, and thus the transitions (δ) of the derived AC-DFA increase more. Accordingly, in order to decrease the number of transition rules, the keywords should be partitioned to small subsets and each subset is processed by a separate string matching engine.

Although the wildcard character is used to reduce the transitions, the evaluation indicates that the number of k-character transitions grows exponentially with respect to k for a k-character AC-DFA. In which, the growth of transitions is approximately 5.7^k for the case of 1,000 keywords by using the proposed AC-DFA approach. Nevertheless, the growth of transitions should be 256^k generally if a k-character transition DFA is implemented in a traditional lookup table approach.

Table 5.3 shows the required spaces of each matching unit and total matching units for different numbers of keywords and different parallel characters k, the required total spaces are derived by multiplying the unit spaces with total rules showed in Table 5.2. In which, b_s is the width of the registers, which is represented in bits. The space of a matching unit is derived by equation (5.2) for an 8-bit character set. The spaces are all represented in bits. This table also lists the numbers of states in the AC-DFAs for different numbers of keywords, and the width (represented as bits) of state registers are denoted in the parentheses. The number of states in an AC-DFA is determined by the keyword set, and further the width of state registers and the total required space are determined according to the number of states. On the other hand, when k is increased, the number of pattern characters and matching outputs and the widths of pattern mask and output mask of the transition rule increase correspondingly. The relationship between the required space and k can be obtained by simply multiplying the number of rules $r_k = len \times p^k$ with the space of a matching unit which is obtained by equation (5.2). The resulting equation is $S_k = len \times p^k \times [(10 + b_s) \times k + 2b_s + 2]$. Consequently, the growth of the required space is about $k \times p^k$.

Table 5.4 shows the results of implementing 4-character AC-DFAs in ASIC devices for the cases of 512 and 1,024 matching units. These implementations were compiled and synthesized by using Altera Quartus II tools with an Altera's HardCopy IV ASIC HC4E35FF1517. According to the data sheet of the product, this

Table 5.3: Spaces for different k-character AC-DFA

Keywords (tot. len.)	States (b_s)	k=1		k=2		k=3		k=4	
		Unit spc.	Tot. spc.	Unit spc.	Tot. spc.	Unit spc.	Tot. space	Unit spc.	Tot. spc.
200 (2,321)	1,790 (11 bits)	45	215K	66	795K	87	2.7M	108	8.2M
400 (4,692)	3,597 (12 bits)	48	769K	70	4.3M	92	21M	114	89M
600 (8,032)	6,081 (13 bits)	51	1.8M	74	13M	97	74M	120	397M
800 (10,728)	8,029 (13 bits)	51	2.7M	74	23M	97	152M	120	937M
1000 (13,374)	9,916 (14 bits)	54	4.0M	78	36M	102	264M	126	1.8G

* spaces are all represented in bits.

Table 5.4: Implementation of 4-character AC-DFA in ASIC

	512 Units	**1,024 Units**
Used HCells	304,666	608,436
HCell Utilization	3%	6%
Total registers	45,432	91,045
Clock	142 MHz	95 MHz
Throughput	4.5 Gbps	3.0 Gbps

ASIC device has 9774,880 HCells, where an HCell is a logic array cell used in the HardCopy IV series devices. In the table, the results include total used HCells, the utilization of HCells and registers, the maximum achievable operating clock, and the derived maximum achievable throughput. The achievable throughput is obtained by multiplying the data width, which is 32 bits, with the clock rate. As of the maximum operating clock, the implementation of 512 matching units is higher than the implementation of 1,024 matching units. The reason for the degradation of operating clock is that the priority multiplexers are implemented by chained multiplexers. The minimal critical path of a priority multiplexer is $log_2 M$ chained multiplexers for M matching units. Therefore, the delay of the priority multiplexer is longer for the more matching units. If the priority multiplexers are re-designed to reduce the time delays, the operating clock should be improved.

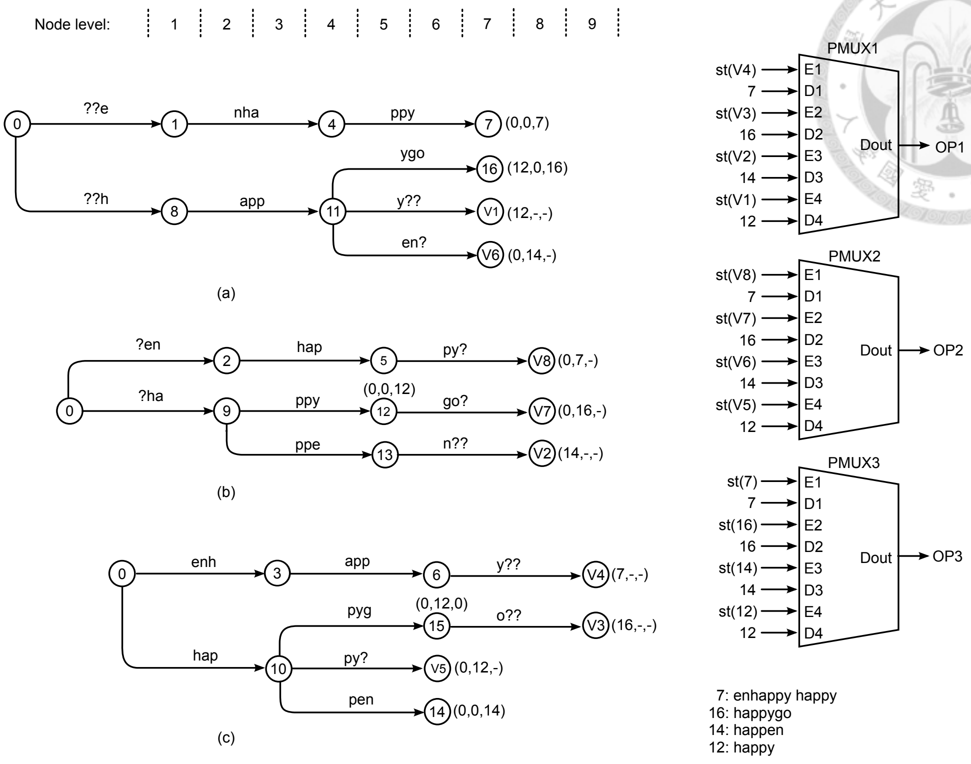


Figure 5.8: Transitions and output multiplexers of the 3-character AC-NFA

5.2 Multi-Character AC-NFA

This section proposes an efficient architecture for implementing the derived multi-character AC-NFA. Fig. 5.8 depicts the 3-character transitions derived from an AC-NFA as three disjoint 3-character AC-NFAs. Although the disjoint 3-character AC-NFAs can be merged to a single 3-character AC-NFA, the disjointed AC-NFAs are explained more clearly. Since the final matching outputs are determined by priority multiplexers and the matching outputs of the nodes in the higher levels have higher priorities, the nodes of the three individual AC-NFAs are arranged according to their levels in the original AC-trie to more clearly illustrate the relationship of the nodes.

For clarity, pseudo nodes are denoted as V1 through V8. The matching outputs that contain non-empty strings are denoted beside the corresponding node. For example, the notation (12,0,16) beside node 16 means that when state 16 is activated the matching outputs corresponding to the three inspecting characters are state 12, 0, and 16, or ‘happy’, an empty string, and ‘happygo’. Level numbers denoted at the top are the levels corresponding to the original AC-trie. Matching outputs OP1 through OP3 are corresponding to the three inspecting characters. The output selection circuit consisting of three priority multiplexers PMUX1 through PMUX3 shown on the right is used to determine the last matching outputs OP1 through OP3 from the matching results of the activated nodes. As describing above, the matching outputs of the nodes in a higher level have higher priorities. The strings represented by matching outputs are depicted in the right lower corner of this figure for clarity.

Because '?' represents an arbitrary character, multiple transitions could be matched simultaneously in the same level. For example, transition δ_3(11,y??)=V1 is always matched when transitions δ_3(11,ygo)=16 is matched. However, transition δ_3(11,y??)=V1 preserves the matching output of pattern 'happy'. Therefore, the matching output corresponding to the first inspecting character can be determined by node V1 alone when these two transitions are matched simultaneously. As another example, when state 15 is activated pseudo state V5 is also activated, where the second matching outputs of node 15 and pseudo node V5 are the same. Therefore, only st(V5) is sent to input E4 of priority multiplexer PMUX1.

Therefore, when multiple transitions in the same level are matched simultaneously, the matching output is determined by the pseudo node that is appended for preserving the matching output for the corresponding inspecting-character. The relationship between the pseudo nodes and the actual nodes can be grouped as follows V1-V5-12, V2-V6-14, V3-V7-16, and V4-V8-7. Where the three members of each group are the matching outputs corresponding to the first through third inspecting characters, respectively. As shown in the figure, the control inputs of PMUX1 and PMUX2 are all the status functions of the pseudo nodes. Consequently, with an increasing number of characters to be inspected in parallel, only the priority multiplexers required for determining the matching outputs must be increased accordingly while the inputs of each priority multiplexer are the same.

The earlier description can be generalized to the case inspecting k characters in parallel, in which the derived k-character transitions can be grouped to k disjoint k-character AC-NFAs. Notably, each AC-NFA is responsible for dealing with a misalignment case. In addition, k priority multiplexers are required for determining the last k matching outputs and each multiplexer has n_{op} inputs.

5.2.1 Implementation

Next, this work describes the logic circuit for implementing the multi-character transitions where the character matching function is implemented by decoders with combinational logics instead of comparators [29]. Moreover, the approach of Sidhu

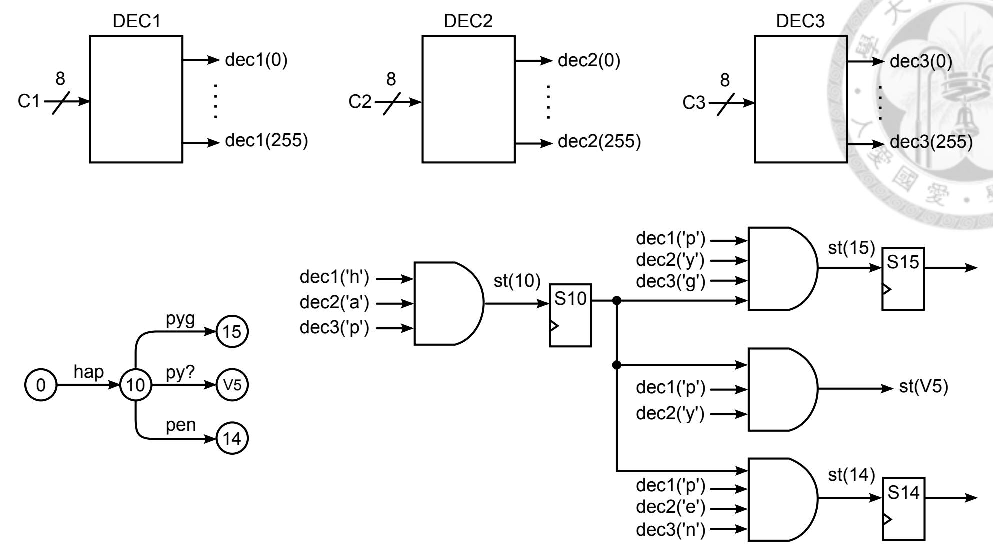


Figure 5.9: Implementation of transitions of the 3-character AC-NFA

and Prasanna [30] that implements the comparators by the Look-Up Tables (LUTs) of FPGAs can be used as well. Fig. 5.9 illustrates an example of the logic circuit of four transitions. In the upper portion of this figure, devices DEC1, DEC2, and DEC3 are 8-to-256 decoders, with each one used to decode one input character to 256 signals. Signals dec1(i), dec2(i), and dec3(i) represent the i-th decoded signals of input characters C1 through C3, respectively, where i is an integer between 0 and 255. For example, when input character C1 is 'e', which corresponds to ASCII code 101, the signal dec1(101) is true. In this figure, the notation dec1('e') instead of dec1(101) is used for clarity.

In addition to depicting the transitions used as the example in the lower left portion, Fig. 5.9 also shows the corresponding logic circuit in the lower right portion. Where signal st(s) denotes the activating signal for the node s, in which s is 10, 14, 15, or V5. Consider a situation in which S10 is true and dec1('p'), dec2('y'), and dec3('g') are true as well. In this situation, signal st(15) is true, implying that transition δ_3(10,pyg)=15 is matched, and node 15 is activated in the next clock.

Alternatively, a multi-character AC-NFA can be implemented in a multi-stage architecture. Nevertheless, it is similar to the implementation of multi-character hybrid AC-FA described later and is omitted here.

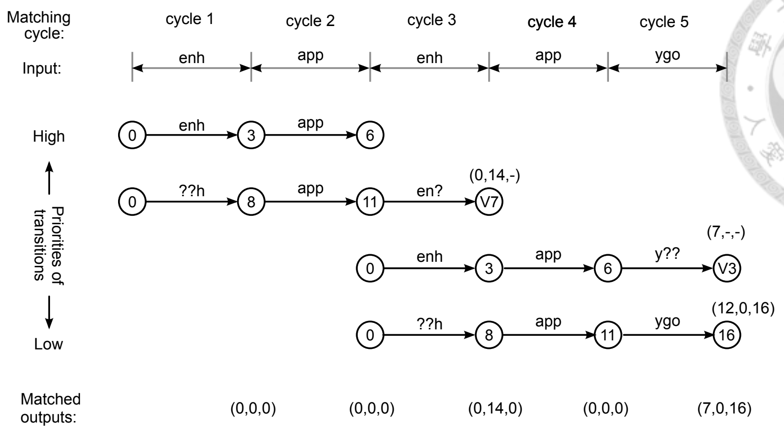


Figure 5.10: Matching example of the 3-character AC-NFA

5.2.2 Matching example

In order to demonstrate the effectiveness of the proposed multi-character AC-NFA approach, Fig. 5.10 illustrates an example to explain the matching operation of the 3-character AC-NFA. In this example, the input string to be matched is 'enhap-penhappygo', which is divided to five 3-character chunks 'enh', 'app', 'enh', 'app', and 'ygo' and then processed in five consecutive matching cycles, respectively. This figure displays only the triggered transitions; the transitions with higher priorities are arranged in the upper portion. The matching results of each matching cycle are depicted in the bottom portion of this figure, which are represented by state numbers for clarity.

Before the matching procedure, all the states are initialized. In the first matching cycle, according to the input characters 'enh', the matched transitions are δ_3(0,enh)=3 and δ_3(0,??h)=8 and both the matching outputs determined by these two triggered transitions are empty strings. In the second matching cycle, according to the input characters 'app' and the next states determined in the previous cycle, the matched transitions are δ_3(3,app)=6, and δ_3(8,app)=11. Both the matching outputs determined by the two triggered transitions are empty strings in the second matching cycle.

In the third matching cycle, according to the input characters 'enh' and the next states determined in the previous matching cycle, the matched transitions are δ_3(11,en?)=V7, δ_3(0,enh)=3, and δ_3(0,??h)=8. The matching result of pseudo node V7 is (0,14,-), in which V7 has a matching output corresponding to the second inspecting character. Consequently, at the end of the third matching cycle, the final matching result OP2 is 14, or 'happen', and the other two results OP1 and

OP3 are both empty strings. In the same way, OP1 through OP3 are all empty strings at the end of the fourth matching cycle. In the fifth matching cycle, two transitions δ_3(6,y??)=V3 and δ_3(11,ygo)=16 are triggered. Both these two triggered transitions have matching results corresponding the first inspecting character which are 7 and 12. Because state 7 is in a higher level than state 12, the final matching result OP1 is determined to be 7. Only δ_3(11,ygo)=16 has valid matching outputs corresponding to the second and third inspecting characters, and which determines the final matching results OP2 and OP3 to be 0 and 16 respectively.

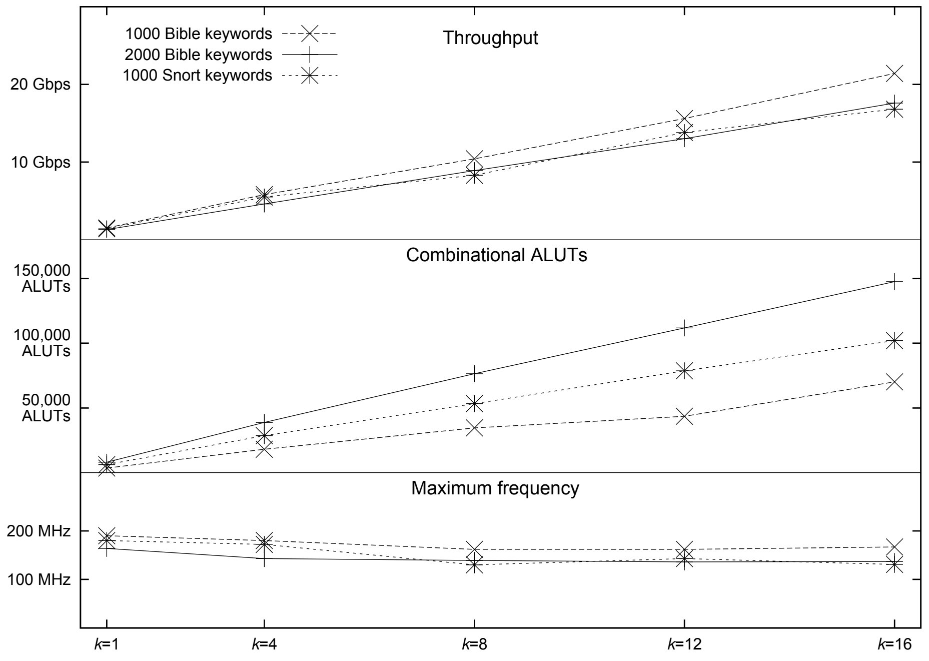


Figure 5.11: Result curves for evaluation of k-character AC-NFAs

5.2.3 Experiment and evaluation

The keywords for evaluation are extracted from the index of the King James Bible and the rules of Snort. The implementations of 1,000 and 2,000 Bible keywords with k = 4, 8, 12, and 16 are evaluated first. For the case of 1,000 Bible keywords, the total length is 7,400 characters and the average length is 7.4 characters; in addition, the AC-trie built on these keywords has 3,982 states, in which 1,125 states are output states. Since the number of goto functions is equal to the number of the states in an AC-trie, the number of the transitions of the k-character AC-NFA is $r_k = 3982 + 1125 * (k - 1)$. For the other case of 2,000 keywords, the total length is 15,414 characters and the average length is 7.7 characters; in addition, the built AC-trie has 8,681 states, in which 2,391 states are output states. Consequently, the number of transitions is $r_k = 8681 + 2391 * (k - 1)$ for the k-character AC-NFA of 2,000 keywords.

A significant amount of research on IDS adopts the Snort rules to evaluate the performance of string matching. Therefore, in this work, Snort keywords are also evaluated for making a comparison with the results of related works. During the evaluation of 1,000 keywords extracted from Snort rules, the total length is 13,566 characters and the average length is 13.6 characters; in addition, the built AC-trie contains 10,157 states and 1,130 output states. Consequently, the number of transitions for k-character AC-NFA is $r_k = 10157 + 1130 * (k - 1)$.

Table 5.5: Evaluation of multi-character AC-NFA with 1,000 Bible Keywords

	$k = 1$	$k = 4$	$k = 8$	$k = 12$	$k = 16$
Total transitions	3,982	7,357	11,857	16,357	20,857
Logic utilization	2%	5%	10%	16%	21%
Used ALUTs	3,583 (< 1%)	18,113 (4%)	34,588 (8%)	53,579 (13%)	70,100 (16%)
Registers	3,106 (< 1%)	3,142 (< 1%)	3,190 (< 1%)	3,238 (< 1%)	3,286 (< 1%)
Fmax (MHz)	190.33	180.18	162.15	162.07	167.36
Throughput	1.5Gbps	5.8Gbps	10.4Gbps	15.6Gbps	21.4Gbps
LE/char	1.13	3.59	6.38	9.60	12.40

Table 5.6: Evaluation of multi-character AC-NFA with 2,000 Bible Keywords

	$k = 1$	$k = 4$	$k = 8$	$k = 12$	$k = 16$
Total transitions	8,681	15,854	25,418	34,982	44,546
Logic utilization	5%	11%	23%	33%	43%
Used ALUTs	8,038 (2%)	38,805 (9%)	76,427 (18%)	111,801 (26%)	147,626 (35%)
Registers	6,902 (2%)	6,944 (2%)	7,000 (2%)	7,056 (2%)	7,112 (2%)
Fmax (MHz)	164.15	142.8	138.52	135.72	137.34
Throughput	1.3Gbps	4.6Gbps	8.9Gbps	13.0Gbps	17.6Gbps
LE/char	1.21	3.71	6.77	9.64	12.55

Programs were developed according to the proposed algorithm to derive the multi-character transitions from the keywords and convert the derived transitions

Table 5.7: Evaluation of multi-character AC-NFA with 1,000 Snort Keywords

	$k = 1$	$k = 4$	$k = 8$	$k = 12$	$k = 16$
Total transitions	10,157	13,547	18,067	22,587	27,107
Logic utilization	4%	8%	16%	24%	31%
Used ALUTs	6,209 (< 1%)	28,589 (7%)	53,386 (13%)	78,715 (19%)	102,030 (24%)
Registers	9,193 (< 1%)	9,235 (2%)	9,291 (2%)	9,347 (2%)	9,403 (2%)
Fmax (MHz)	180.28	172.47	130.17	143.39	131.27
Throughput	1.4Gbps	5.5Gbps	8.3Gbps	13.8Gbps	16.8Gbps

to VHDL codes. The generated VHDL codes were compiled and synthesized by Altera's development tool Quartus II 9.1. The hardware function was verified by ModelSim-Altera 6.5b software. The device selected for evaluating the proposed architecture is an Altera's Stratix IV family FPGA EP4SE530F43C2 which has 424,960 ALUTs and 424,960 Registers, where an ALUT (adaptive look-up table) is a logic unit used in the Altera's FPGA devices. The achievable throughput is derived by multiplying the data width with the maximum frequency (F_{max}) reported by the development tool.

Tables 5.5 and 5.6 summarize the evaluation results for 1,000 and 2,000 Bible keywords, respectively. In addition, Table 5.7 lists the evaluation results for 1,000 Snort keywords. Fig. 5.11 shows the curves of the hardware costs, the maximum frequencies, and derived throughputs for the implementations with respect to different k values. The curves reveal that the throughputs and hardware costs are linearly proportional with respect to k, whereas the maximum frequencies (F_{max}) are slightly decayed as k increases.

Tables 5.5 and 5.6 indicate that the throughputs grow about 14 times, and the used ALUTs grow about 18 times for $k = 16$ with respect to $k = 1$ for both cases. In the case of 1,000 Snort keywords, the throughputs grow about 12 times and the used ALUTs grow about 16 times for $k = 16$ with respect to$k = 1$, as shown in Table 5.7. In these implementations, the used ALUTs of the hardware resource grows nearly linearly with respect to k, whereas the used registers remain nearly the same since the states do not increase as k increases and the pseudo states are not saved.

Analysis results indicate that the F_{max} is degraded slightly as k increases. The max frequencies F_{max} in the case of 2,000 Bible keywords are always lower than in the case of 1,000 Bible keywords for different k values, even though the former has fewer transitions. For example, the used ALUTs and F_{max} in the case of 1,000 keywords with $k = 16$ are 70,100 ALUTs and 167.36 MHz, and those in the cases of 2,000 keywords with $k = 1$ and $k = 4$ are 8,038 ALUTs and 164.15 MHz and 38,805 ALUTs and 142.8 MHz respectively, where the first one has more transitions while can achieve a higher F_{max}. This phenomenon might be owing to the fact that the critical path of the circuit is dominated by the complexity of output-selection circuit that depends on the number of output states (n_{op}). Routing between the output states and the output selection circuit becomes more complex as n_{op} increases, whereas the routing complexity increases slightly as k increases. Moreover, max frequencies F_{max} in the case of 1,000 Snort keywords are lower than that in the case of 1,000 Bible keywords. This difference might me owing to the fact that the average length of the Snort keyword is nearly twice that of the average length of the Bible keyword.

In these implementations, the priority multiplexers consist of multiple levels of multiplexers, and the critical path is the delay produced by $\log_2 n_{op}$ levels of multiplexers for n_{op} output states, explaining why the delay of the priority multiplexers is longer for the more inputs. In the proposed approach, the number of inputs for a priority multiplexer is equal to the number of output states. In the case of 1,000 Bible keywords, 1,125 output states exist and the critical path of the priority multiplexer is 11 levels of multiplexers according to $\log_2 1125 = 10.13$. In the case of 2,000 Bible keywords, 3,982 output states exist and the critical path of the priority multiplexer is 12 levels of multiplexers according to $\log_2 2391 = 11.22$. In the case of 1,000 Snort keywords, 1,130 output states exist and the critical path of the priority multiplexer is 10 levels of multiplexers according to $\log_2 1130 = 10.14$.

Table 5.8: Evaluation with 1,000 Snort Keywords for Matching-Flag Output

	$k = 1$	$k = 4$	$k = 8$	$k = 12$	$k = 16$
Total transitions	10,157	13,547	18,067	22,587	27,107
Logic utilization	7%	9%	16%	24%	33%
Registers	1,701 (1%)	10,105 (7%)	15,372 (11%)	22,751 (16%)	30.226 (21%)
Registers	8,682 (6%)	8,195 (6%)	7,804 (6%)	7,841 (6%)	7,860 (6%)
Fmax (MHz)	333.44	280.5	245.94	203.79	167.11
Throughput	2.7Gbps	9.0Gbps	15.7Gbps	20.0Gbps	21.4Gbps
LE/char	0.96	1.69	2.14	2.82	3.51

In some applications (e.g., the string-matching circuit for IDS described in the article of Katashita et al. [18], in which only the matching output for each pattern is required), the priority multiplexers can be replaced with OR-gates. This work also examines how the priority multiplexers influence performance by undertaking additional experiments with the architecture that determines the matching outputs by OR-gates instead of priority multiplexers. During this evaluation, the implementation is built on a smaller FPGA EP4SGX180DF29C2X, which has 140,600 ALUTs and 140,600 registers. Table 5.8 summarizes the evaluation results with only matching flags as output. Comparing Table III with Table IV reveals that when the matching outputs are determined by priority multiplexers, the required ALUTs increase approximately 3.36 times. Additionally, Fmax decreases by approximately 38%, implying that the critical path delay increases on average by 1.61 times.

In Katashita et al. [18], the hardware resources required for every character are 0.83 LE/char and 4.13 LE/char for the cases of 1-character and 16-character NFAs, respectively. In the proposed approach, the hardware costs are 0.96 LE/char and 3.51 LE/char for the cases of 1-character and 16-character NFAs, respectively, when only matching flags are output, which resembles the results of Katashita et al. When the output stage is implemented by priority multiplexers, the hardware costs increase to 1.42 LE/char and 10.27 LE/char for the cases of 1-character and 16-character NFAs, respectively. Despite the increases in hardware cost and time delay when using priority multiplexers to determine the matching outputs, providing a corresponding matching output represented in a state number for each inspected character should be convenient in most applications.

5.3 Multi-Character Hybrid AC-FA

This section proposes a multi-stage string-matching architecture for implementing the derived multi-character hybrid AC-FA. Fig. 5.12 illustrates the transitions of the 3-character hybrid AC-FA. The transitions of NFA portion and DFA portion are separated for clarity. Comparing with the 3-character AC-NFA, the 3-character hybrid AC-FA has three additional transitions δ_3(7,go?)=-, δ_3(6,ygo)=16, and δ_3(6,en?)=- in the DFA portion.

5.3.1 Implementation

Fig. 5.13 illustrates the block diagram of the proposed architecture which implements a 3-character hybrid AC-FA, which derived from the example of Fig.3.11.

The number of the stages L is determined by the number of NFA levels N_{NFA}

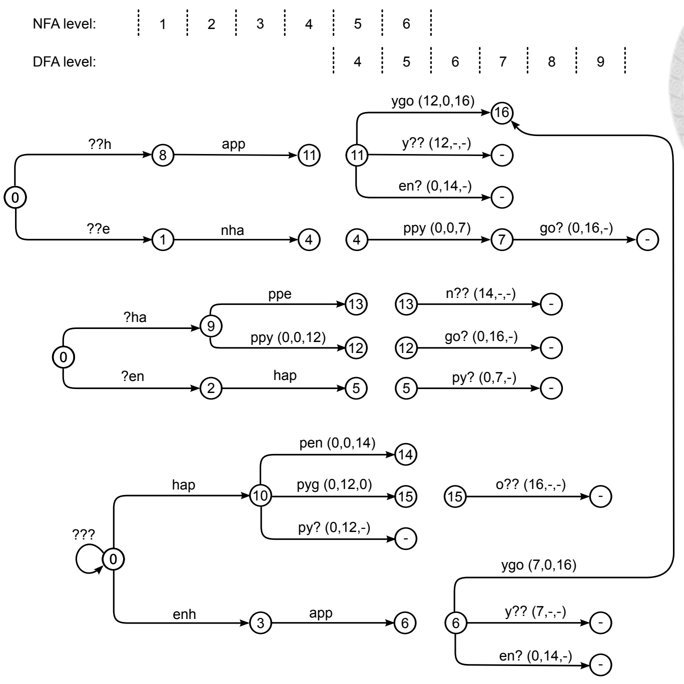


Figure 5.12: Transitions of multi-character hybrid AC-FA

and k value, that is $L = N_{NFA} + k + 1$. For the described example, the NFA-level is $N_{NFA} = 3$ and the number of characters under inspection in parallel is $k = 3$, so that the number of stages $L = 3 + 3 + 1 = 7$.

The first six stages are arranged as three chains where each stage chain deals with a alignment case. Stage 1, the first stage of the first chain, deals with the case that the first character of pattern appears in the third character of the input chunk, and therefore the preceding two characters of the transitions are both wildcard characters.

The transitions in stage 7, which is a terminal stage, are corresponding to the DFA portion of the hybrid AC-FA. The next states determined by the final stages, which are stages 4 through 6, in the three chains all traverse into stage 7. Because at most one state can be activated in a time for each stage, the priority multiplexer PMUX0 determines the next state for stage 7 from the next states (NXs) output from stages 4 through 7. The transitions in the later stage are corresponding to the states in the deeper level of the AC-trie, and thus the next state determined by the later stage has higher priority, e.g. the next state determined by stage 7, which is the final stage, has the highest priority.

Since multiple transitions may be triggered simultaneously, the transitions in each stage are arranged according their priorities where a transition with a higher

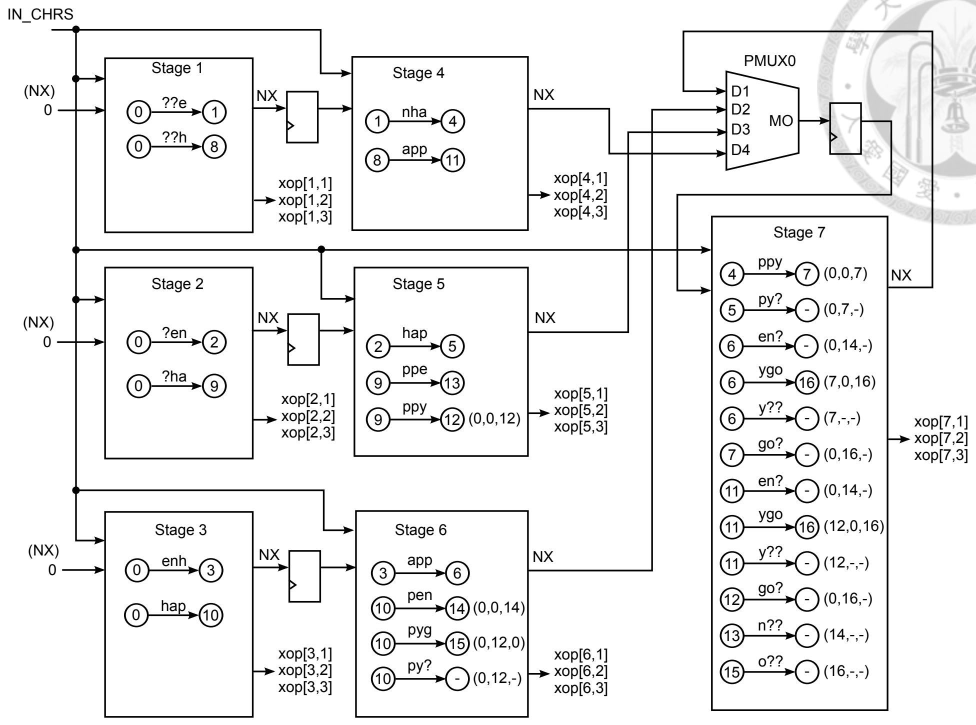


Figure 5.13: Implementation of 3-character Hybrid AC-FA

priority is arranged in a higher position in the block diagram. For example, transition δ_3(6,y??)=- is triggered always when the transition δ_3(6,ygo)=16 is triggered, and thus the former should have a lower priority. However, if two transitions are never triggered at the same time then their priorities do not matter.

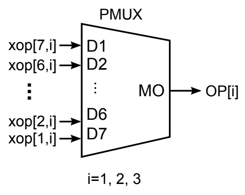


Figure 5.14: Matching output circuit

In the multi-stage hybrid-FA approaches, the matching output and next state are determined by using priority multiplexers from multiple potential results. Fig. 5.14 illustrates the priority multiplexer for determining the i-th final matching output corresponding to the i-th inspecting character. Three priority multiplexers are required for determining three matching outputs in the above example. A priority multiplexer accepts multiple inputs and selects a valid input with the highest priority as the output; if none of the inputs is valid then a default value is output, such as zero in general. Where the input in the upper position has a higher priority, for example, the input D1 has the highest priority.

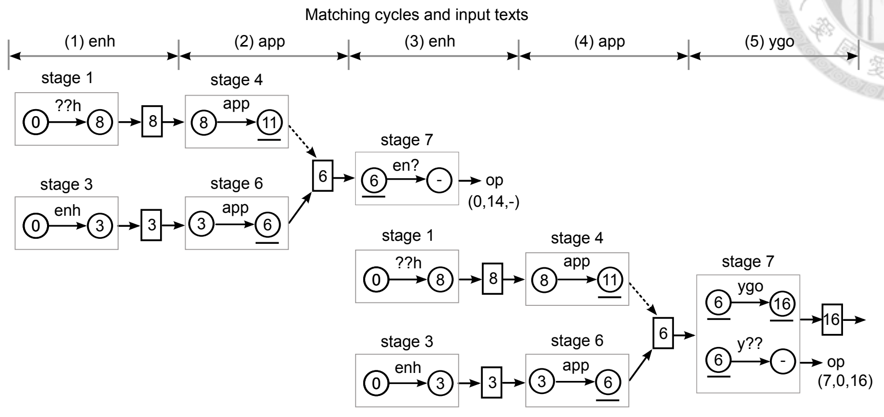


Figure 5.15: Matching example of the 3-character hybrid AC-FA.

5.3.2 Matching example

In order to demonstrate the effectiveness of the proposed approach, Fig. 5.15 illustrates an example to explain the matching process of the 3-character hybrid AC-FA. The input text of this example is 'enhappenhappygo', which is split to chunks of three characters and then processed sequentially in five matching cycles. In the first cycle, according to the input 'enh', the transitions δ_3(0,??h)=8 in stage 1 and δ_3(0,enh)=3 in stage 3 are triggered. Neither the transition has matching output so the matching outputs are all empty in this cycle. The next states determined by stages 1 and 3 are sent to stages 4 and 6, respectively.

In the second cycle, according to the input 'app' and the next states determined in preceding cycle, the transitions δ_3(8,app)=1 in stage 4 and δ_3(3,app)=6 in stage 6 are triggered which determine next states 11 and 6, respectively. Both states 11 and 6 are in DFA portion and the latter one is in deeper depth, so that state 6 is preserved and sent to stage 7. The matching outputs are all empty in this cycle also.

In the third cycle, according to the input 'enh' and the next states determined in preceding cycle, the transitions δ(6,en?)=- in stage 7, δ(0,??h)=8 in stage 1, and δ(0,enh)=3 in stage 3 are triggered. The transitions δ(6,en?)=- determines the matching output corresponding to the second input character to be 14 or 'happen'.

In the fourth matching cycle, transitions δ_3(8,app)=1 in stage 4 and δ_3(3,app)=6 in stage 6 are triggered in response to the input characters 'app' and the next states determined in the preceding matching cycle. The priority multiplexer PMUX0 selects the next state determined by the stage 6, which is 6, to sent to stage 7. The matching outputs are all empty strings in this cycle.

In the fifth matching cycle, two transitions of stage 7 which are δ_3(6,ygo)=16 and δ_3(6,y??)=- are triggered in response to the input characters 'ygo' and the next states determined in the preceding matching cycle. The transition δ_3(6,ygo)=16 has a higher priority so that it determines the next state to be 16 and the matching outputs corresponding to the three input characters to be 7, 0, and 16 in this matching cycle. The first and third matching outputs, 7 and 16, are corresponding to keywords 'enhappy happy' and 'happygo' respectively.

5.3.3 Estimation of required stages

This subsection estimates the required stages in a multi-stage string matching engine from two perspectives. First, in order to understand how many stages are required to effectively identity the keywords if a multi-stage string-matching engine only has the NFA portion, the probabilities of sharing common prefixes of keywords for different lengths are examined. Alternatively, the number of multi-character transitions versus the levels of the NFA-portion is evaluated to estimate the total stages are required in the hybrid AC-FA architecture.

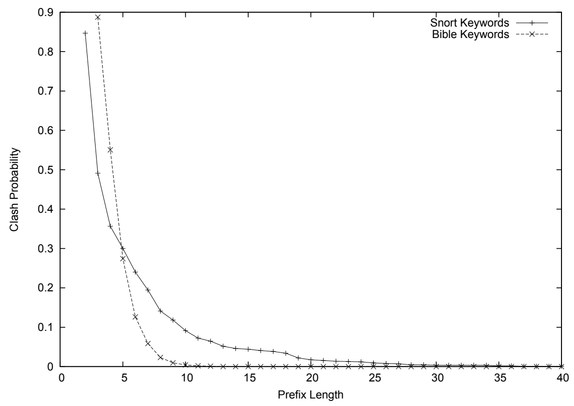


Figure 5.16: The probability distribution of sharing common prefixes of various lengths

First, the probabilities of sharing common prefixes of keywords for different lengths are examined to understand how many stages are required if a multi-stage string-matching engine only has the NFA portion. Figure 5.16 illustrates the graph of analysis results. The keywords for analysis are extracted from the index of the

King James Bible and the rules of Snort. The average lengths are 8 and 22 characters and the longest keywords are 37 and 80 characters for the Bible and Snort keywords, respectively.

Because many keywords of the Bible keywords have common prefixes, such as ‘ABEL’, ‘ABEL-BETH-MAACHAH’, ‘ABEL-MEHOLAH’, ‘ABEL-MIZRAIM’, and ‘ABEL-SHITTIM’, the probabilities of baring common prefixes is high for the cases the lengths of prefixes are less than five characters. As shown in Figure 5.16, the probability for sharing the 10-character common prefixes of the Bible keywords is 0.09. While for the Snort keywords, it needs to share 25-character common prefixes to achieve the probability of 0.09. As a result, to assure that the probability is low enough, the length of prefixes to be compared in the NFA portion should be larger than the average length of keywords, which is 22 for Snort keywords.

Next, to evaluate the number of multi-character transitions versus the levels of the NFA-portion, the samples for evaluation are 1,000 keywords retrieved randomly from SNORT [31] rules. The numbers of k-character transitions are evaluated for the cases of k=1, 4, and 8, and NFA levels are 1 to 15, respectively. Fig. 5.17 shows the relation between the numbers of transitions and levels of NFA-portion. The AC-trie built on these keywords has 10,157 goto functions and 1,130 non-empty outputs, so that the number of transitions is $10,157 + (k-1) * 1,130$ for a k-character AC-NFA. The numbers of transitions of the k-character AC-NFAs are denoted as horizontal dashed lines in this figure.

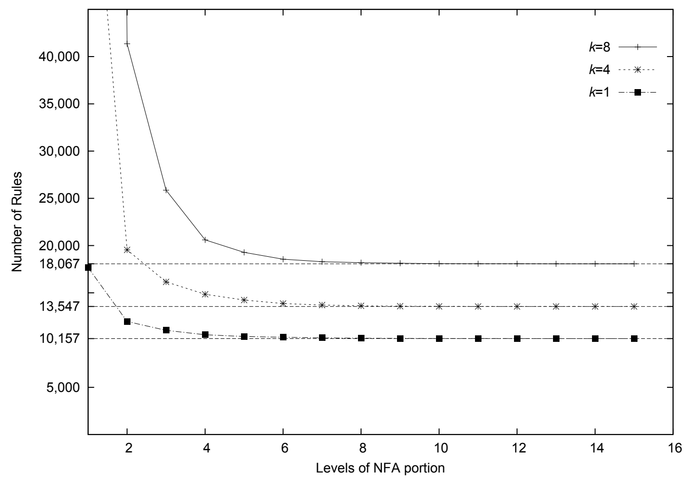


Figure 5.17: Transitions versus NFA-levels for the k-character Hybrid AC-FA

As seen in this graph, the transitions increase dramatically when the NFA levels are less than four; while the curves are nearly flat when the NFA levels are greater than eight. Moreover, when the NFA levels are more than 15, the number of transitions of the k-character hybrid AC-FA is equal to that of the k-character AC-NFA. This comparison indicates that the transitions grow almost linearly with respect to k as the NFA levels are more than eight. As can be seen in this result, the benefit of the hybrid approach is more significant when the k is larger. Although the longest keyword has 80 characters, the transitions do not increase when the NFA levels are more than 15. In other words, eighty stages are required if the matching engine is implemented in an AC-NFA approach.

Table 5.9: Implementing multi-character hybrid AC-FAs on FPGA

	k=1	k=4	k=8
Total rules	2,813	3,756	5,013
Used ALUTs	11,741 (3%)	21,460 (5%)	31,258 (7%)
Used registers	132 (<1%)	204 (<1%)	300 (<1%)
Logic utilization	3 %	6 %	9 %
Max. Freq.	83.93 MHz	76.44 MHz	66.6 MHz
Throughput	0.7 Gbps	2.4 Gbps	4.3 Gbps
LE/char	3.81	6.96	10.14

5.3.4 Experiment and evaluation

The proposed architecture of multi-character hybrid AC-FA was evaluated on FPGA devices. The proposed architectures were designed in VHDL code and then built by Altera's development tool Quartus II. The built results were simulated in Modelsim to verified the logic functions of the architectures. The devices selected for evaluating the proposed architecture is Altera's products Stratix IV FPGA, which has 424,960 ALUTs and 424,960 registers. The proposed implementation was evaluated with 300 keywords and the results are shown in Table 5.9. The total length of the 300 keywords is 3,892 characters. Comparing the results of k=4 and 8 with that of k=1 shows that the throughputs grow 3.4 and 6.1 times, respectively, while the used ALUTs grow 1.8 and 2.7 times, respectively. The growth rates of used ALUTs are lower than those of throughputs. Comparing the results also indicates that the maximum operating clock is degraded when the transitions grow. The reason for clock degradation as transitions growing is that the minimal critical path of a priority multiplexer with M inputs consists of $log_2 M$ chained multiplexers; therefore the delay of the priority multiplexers is longer for the more inputs.

Chapter 6

Configurable Architectures

This chapter presents configurable architectures of multi-character string-matching. Although the hybrid-FA approach has the space efficiency and the stages of which can be determined as required, the number of rules in each stage is varied with the keyword set to be processed. Therefore, a configurable architecture consisting of multiple rule units is proposed, in which each rule unit can be configured to process a specific transition and be allocated to a specific stage. Due to the configurability features, the proposed architectures can process different keyword sets by simply updating the configuration.

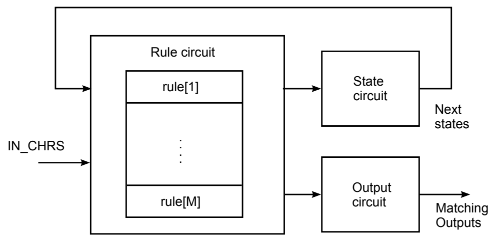


Figure 6.1: Main block diagram of the configurable architecture

6.1 Configurable stage architecture

This section describes a architecture that includes multiple configurable rule units each of which can be dynamically configured to a desired stage. The proposed configurable architecture can be applied to the AC-DFA or hybrid-FA.

6.1.1 Block diagram

Fig. 6.1 illustrates the block diagram of the proposed configurable architecture, which includes rule, state, and output circuits. All rule units are put together in the rule circuit in which each rule unit processes a transition. A rule unit can be allocated to a desired stage according to its own rule data. The state circuit determines the next states of stages; and the determined next states are looped back to the rule circuit as current states in the next matching cycle. The output circuit determines the final matching outputs corresponding to the inspecting characters from the matching results of all rule units.

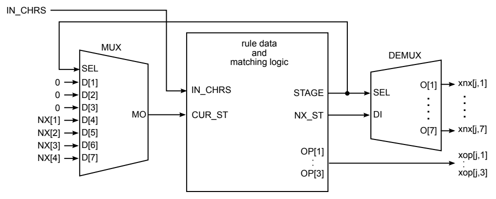


Figure 6.2: Rule unit for the configurable stage architecture

Fig. 6.2 illustrates the block diagram of a rule unit. The rule data of each rule unit contains stage information used to determine which stage this rule unit belongs to. According to the stage information, the multiplexer selects the corresponding input state and the demultiplexer sends the resulting next state to the corresponding output.

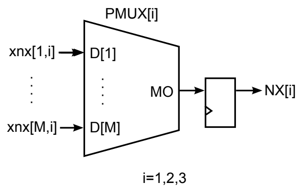


Figure 6.3: State circuit of stages 1 through 3 for the configurable stage architecture

The circuits for determining the next states NX[1] through NX[3] are identical, each of which is implemented by a priority multiplexer as shown in Fig. 6.3. The circuit for determining the next state NX[4] consists of two levels of priority multiplexers, as shown in Fig. 6.4. The priority multiplexers PMUX4 through PMUX7 determine the next states from the matching results corresponding to the stages 4 through 7 and then PMUX8 determines NX[4] from the results of PMUX4 through PMUX7. The output circuit consists of multiple priority multiplexers each of which

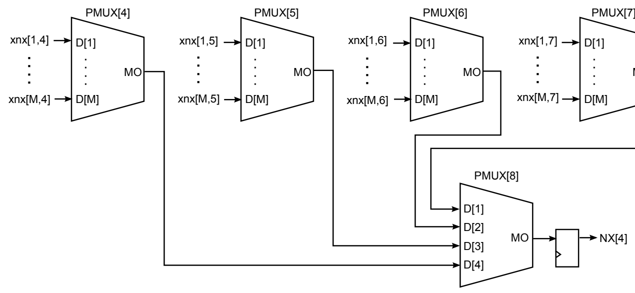


Figure 6.4: State circuit of the terminal stage for the configurable stage architecture

is similar to the block diagram shown in Fig. 5.14, while the priority multiplexers must have M inputs according to the M rule units here. Therefore the detail of the output circuit is omitted here.

The proposed configurable matching engine can be manufactured as a standalone chip. Consequently, when the processing keyword set is changed, only the transitions need to be regenerated according to the new keyword set, and then the rule units are reconfigured by using the new generating transitions. Because of the flexibility, the proposed configurable matching engine can be implemented in ASICs and not restricted in FPGAs. However, there is a trade off between the flexibility and performance. While the proposed configurable matching engine is flexible, the circuit becomes complicated and the performance is degraded. However, this degradation in performance can be compensated by advanced semiconductor technologies.

6.1.2 Experiment and evaluation

The proposed architectures are evaluated on FPGA and ASIC devices. The proposed architectures were designed in VHDL code and then built by Altera's development tool Quartus II. The built results were simulated in Modelsim to verified the logic functions of the architectures. The devices selected for evaluating the proposed architectures are Altera's products Stratix IV FPGA and HardCopy IV ASIC, where the FPGA has 424,960 ALUTs and 424,960 registers and the ASIC device has 9,774,880 HCells.

Since the multi-stage architecture is implemented by decoders and combination logics instead of registers and comparators [29] and must be rebuilt when the keyword set is changed, it is only evaluated on FPGA devices. While the rule data of the configurable architecture can be reconfigured when the keyword set is changed, it is evaluated on both FPGA and ASIC devices. The NFA levels are 8 for all of the implementations, and thus the stages are 10, 13 and 17 for $k = 1$, 4, and 8, respectively. Additionally, the achievable throughput is derived by multiplying the data width with the clock rate.

Table 6.1: Implementing the configurable architecture on FPGA

	Original			**2-stage pipeline**		
	k=1	k=4	k=8	k=1	k=4	k=8
Used ALUTs	60,119 (14%)	77,948 (18%)	106,632 (25%)	59,544 (14%)	79,033 (19%)	102,315 (24%)
Used registers	27,063 (6%)	60,921 (14%)	106,532 (25%)	35,090 (8%)	72,580 (17%)	120,749 (28%)
Logic utilization	24 %	34 %	48 %	24 %	34 %	47 %
Max. Freq. (MHz)	34.34	25.97	20.54	74.75	57.95	48.44
Throughput (Gbps)	0.3	0.8	1.3	0.6	1.9	3.1

The proposed configurable architecture is evaluated on FPGA and ASIC devices. These implementations consist of 512 rule units for the purpose of verification and evaluation. Moreover, the pipeline approach of Soewito [32], which integrates the pipeline architecture and multi-thread operation, is applied to increase the throughput in the evaluation of configurable architecture. The pipeline configurable architecture for evaluation includes two stages.

Table 6.1 summarizes the results of implementing the original and pipeline configurable architectures on FPGAs. The results of implementing the original architecture on FPGAs are discussed first. The derived throughputs grow approximately 2.7 and 4.3 times for k=4 and 8 with respect to k=1, respectively. Considering the hardware resources, the used ALUTs grow 1.3 and 1.8 times, and the used registers grow 2.3 and 3.9 times for k=4 and 8 with respect to k=1, respectively. Next, the results of implementations of the pipeline architecture on FPGAs are discussed. The derived throughputs grow approximately 3.2 and 5.2 times for k=4 and 8 with respect to k=1, respectively. Considering the hardware resources, the used ALUTs grow 1.3 and 1.7 times, and the used registers grow 2.1 and 3.4 times for k=4 and 8 with respect to k=1, respectively.

Subsequently, Table 6.2 summarizes the results of implementing the original and pipeline configurable architectures on ASICs. The results of implementing the original architecture on ASICs are discussed first. The derived throughputs grow approximately 3.8 and 5.8 times for k=4 and 8 with respect to k=1, respectively. Whereas the used HCells grow 1.6 and 2.5 times for k=4 and 8 with respect to k=1, respectively. Next, the results of implementing the pipeline architecture on ASICs

Table 6.2: Implementing the configurable architecture on ASIC

	Original			**2-stage pipeline**		
	k=1	k=4	k=8	k=1	k=4	k=8
Used HCells	389,094	626,943	966,938	383,247	629,855	943,581
HCell utilization	4 %	6 %	10 %	4 %	6 %	10 %
Max. Freq. (MHz)	77.66	70.76	54.18	172.0	136.46	123.44
Throughput (Gbps)	0.6	2.3	3.5	1.4	4.4	7.9

are discussed. The derived throughputs grow approximately 3.1 and 5.6 times for k=4 and 8 with respect to k=1, respectively. Additionally, the used HCells grow 1.6 and 2.5 times for k=4 and 8 with respect to k=1, respectively.

For the implementations on ASIC, the maximum operating clock for k=4 is degraded 9% as comparing with that for k=1, while the maximum operating clock for k=8 is degraded 23% as comparing with that for k=4. The reason of the clock degradation much more for k=8 might be because the multiplexers and demultiplexers used in the implementation are 32 to 1 for k=8, while are 16 to 1 for k=1 and k=4.

Comparing Table 5.9 with Table 6.1 indicates that the maximum frequencies of the configurable architecture are lower than those of the multi-stage architecture for the implementations on FPGAs. This might be because the configurable architecture is more complicated than the multi-stage architecture. Furthermore, since the rule matching function is implemented by decoders with combinational logics instead of registers and comparators in the implementations of the multi-stage architecture, its hardware efficiency and achievable clock rate are much higher than the implementations of the configurable architecture. Both in the implementations on FPGA and ASIC, when k increases the hardware utilization is increased and the maximum operating clock is degraded, while the implementations on ASIC have higher clock rates and hardware efficiency. These results imply that the multi-stage architecture should be used if FPGA is selected for an application, while the configurable architecture should be the choice if ASIC is selected or a stand alone chip is going to be designed. Tables 6.1 and 6.2 also indicate that the throughput can increase approximately twice by using a two-stage pipeline structure, whereas the used hardware resources increase slightly.

6.2 Configurable data-width architecture

This section proposes an architecture of configurable rule unit to provide the flexibility of the multi-stage architecture. The rule units can be configured according to the keyword set to be processed. In addition, multiple rule units can be grouped together to process more characters in parallel, i.e., the number of characters to be inspected in parallel is configurable. For example, a rule unit can process two characters in parallel, and then a unit-group consisting of four units can process eight characters in parallel.

Furthermore, in the proposed configurable architecture, all rule units are put together and each one can be allocated to a different stage dynamically according to the rule data. Consequently, this configurable architecture can process various keyword sets by simply reconfiguring the rule data.

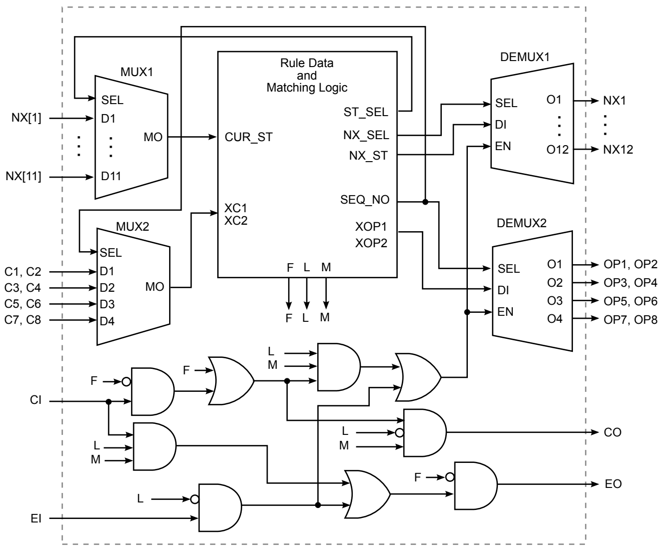


Figure 6.5: Rule unit for the configurable data-width architecture

6.2.1 Basic rule unit

Figure 6.5 illustrates the block diagram of a basic rule-unit, and the table in Figure 6.6 briefly lists the symbol descriptions for the rule unit. The configurable settings of a basic rule-unit include ST_SEL, NX_SEL, F, L, CUR_ST, C1, C2, NX_ST, OP1, and OP2. Notably, some symbol names of the configurable settings are same as the signal names for simplicity.

The multiplexer MUX1 selects the corresponding state to the input CUR_ST

Symbol	Description
SEQ_NO	Sequence no.
ST_SEL	Current state selection
NX_SEL	Next state selection
F	First unit flag
L	Final unit flag
M	Matching result
EN	Output enable
CI	Carry input
CO	Carry output
EI	Enable input
EO	Enable output

Figure 6.6: Symbol descriptions of the configurable data-width rule-unit

according the setting ST_SEL. The demultiplexer DEMUX1 sends the resulting next state NX_ST to the corresponding output according to the setting NX_SEL. The settings ST_SEL and NX_SEL are determined by which stage that the unit belongs to. In this example, MUX is a 12-to-1 multiplexer and DEMUX is a 1-to-12 demultiplexer, the basic rule-unit can be configured to stage 1 to 12. However, since the twelfth stage is the final stage, the twelfth input of MUX1 is omitted.

The multiplexer MUX2 selects corresponding characters to the inputs XC1 and XC2 according the setting SEQ_NO. The demultiplexer DEMUX2 sends OP1 and OP2 to corresponding matching outputs according to the setting SEQ_NO.

Multiple rule units can be cascaded as a unit-group through control signals CI, CO, EI, and EO. The signals CI and CO are carry signals; and EI and EO are enable signals. The basic rule-unit in this example can inspect two characters in parallel. Consequently, a unit-group consisting of n rule units can inspect $2n$ characters in parallel. In a unit group, the matching results of rule units are aggregated at the final unit through CI and CO. In which, the signal L is true for the final unit of a group. The final matching result obtained in the final unit are feed back to other units through signals EI and EO.

Signals F and L determine the carry and enable signals should be propagated to another unit or not. Signal F is a first-unit flag. When a unit is the first unit of a unit-group, the signal F of which should be true. Signal L is a final-unit flag. When a unit is the final unit of a unit-group, the signal L of which should be true.

The table in Figure 6.7 summarizes the decisions for determining the control signals EN, CO, and EO. In the first three cases, a group consists of multiple cascaded

Flag		Control signal			Description
F	**L**	**CO**	**EO**	**EN**	
0	0	CI∧M	EI	EI	middle unit
0	1	0	CI∧M	CI∧M	final unit
1	0	M	0	EI	first unit
1	1	0	0	M	standalone

Figure 6.7: Decisions of control signals for the configurable data-width rule-unit

units; while in the last case, a group has only one unit.

The first case is that a unit is in the middle of multiple cascaded units, i.e. both F and L are configured as false. A middle unit must aggregates the value received from CI with its own matching result M and then forwards the aggregating result to the next unit, so that CO is obtained by CI∧M. Fur a middle unit, the enable signal EN is equal to the value received from EI, and the value of EI is passed to the previous unit through EO.

The second case is happened in the final unit of a group. Since there is no other unit after the final unit the signal CO should be false. The final unit is responsible for aggregating the matching result of a group that must consider the value of CI and its own matching result M, so that EN is obtained by CI∧M. The final unit also must feed the obtained EN backward to the previous unit through output EO.

The third case is considered for the first unit of a group. There is no other unit before the first unit and thus the signal EO is false. In addition, the output CO of the first unit is simply equal to its own matching result M. The enable signal EN of the first unit is equal to the value of EI directly. A unit as described in the last case is a standalone unit, i.e. a single unit processes a transition alone. The signals CO and EO are not used in a standalone unit and thus both CO and EO are always false. The signal EN of a standalone unit is simply equal to its own matching result M.

Since each unit-group needs to generate only one next-state, only one rule unit is required to generate the next state when this unit-group is matched. While every rule unit has to produce the matching outputs corresponding to the characters inspected by that unit when a unit-group is matched.

Considering that each basic rule-unit can process a 2-character transition and the requirement is to process 6-character transitions and wants to have total 12 stages. In this scenario, each unit-group should consist of three basic units. The resulting architecture has six stage-chains and each stage-chain consists of two stages.

According to this configuration, the longest pattern that can be processed is varied from 7 to 12 characters corresponding to the first to sixth stage-chain respectively. As a result, the configurable data-width allows a user to choose the required configuration depend on the application. In addition, the characters inspected in parallel are more, the throughput gains more.

According to the proposed configurable approach, when the processing keyword set is changed, only the transitions need to be regenerated for the new keyword set, and then the rule units are reconfigured by using the new generating transitions. Because of the flexibility, the proposed configurable matching engine can be manufactured as a standalone device, for example, manufactured by using ASICs. However, there is a trade off between the flexibility and performance. While the proposed configurable matching engine is flexible for a real application, the circuit becomes complicated and the performance is degraded. Nevertheless, this degradation in performance can be resolved by advanced semiconductor technologies.

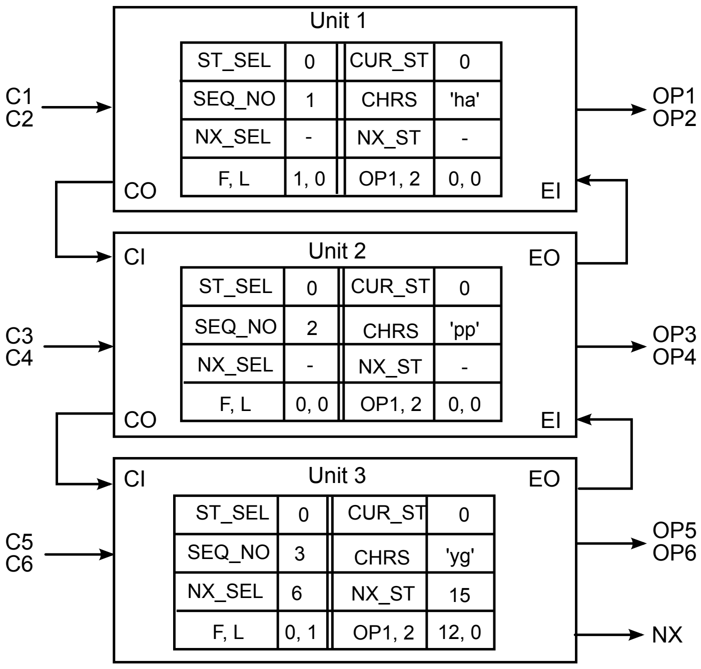


Figure 6.8: Example of three-unit group

6.2.2 Example of three-unit group

Figure 6.8 illustrates an example of three concatenated rule-unit configured as a 6-character transition. In this figure, only the used signals are shown for clarity. The transition represented by this example is δ_6(0,happyg)=15. In which, the matched output OP5, corresponding to the fifth inspecting character C5, is 12.

The first unit, the rule data SEQ_NO of which is 1, selects characters C1 and C2 as input characters. The flag F being 1 indicates that no other unit is before this unit, and the flag L being 0 indicates that it has a successive unit. In the second unit, SEQ_NO is 2 and the flags F and L of which are both 0. The first and second units do not output next states. Only the third unit, which is the final unit of this group, output a next state NX.

As described above, the number of transitions grows linearly with respect to k for a k-character AC-NFA. Furthermore, the space required to store a k-character rule also grows linearly with respect to k approximately. Consequently, the required space grows quadratically with respect to k for a k-character AC-NFA implemented in the proposed multi-stage architecture. In short, the space complexity of the proposed configurable multi-character multi-stage string-matching architecture is $O(n^2)$ with respect to the number of inspecting characters.

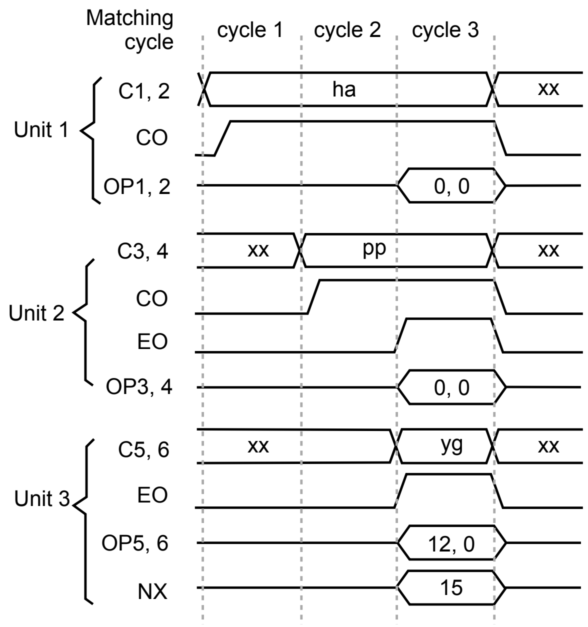


Figure 6.9: Waveform of the three-unit group

6.2.3 Waveform of three-unit group

Fig. 6.9 illustrates the waveform of the matching operations of the example shown in Fig. 6.8. Because the start state of the example transition is 0, i.e. the root node, and does not need to be checked in matching operations, the CUR_ST is not shown in this waveform diagram. For clarity, the input characters and matching outputs are denoted in groups. For example, the input characters of unit 1 are denoted as C1,2 and the matching outputs of which are denoted as OP1,2. In addition, the propagation delays in circuit are not considered in this waveform diagram.

In matching cycle 1, according to the input characters ‘haxxxx’, only unit 1 is matched and its CO becomes true. In matching cycle 2, according to the input characters ‘happxx’, units 1 and 2 are matched and their CO become true. In matching cycle 3, according to the input characters ‘happyg’, all units of this group are matched and the matching result aggregated in unit 3 is true. The matching result obtained in unit 3 is fed backward to units 2 and 1 through terminals EI and EO, and thus all matching results are enabled to be output together. After matching cycle 3, the input characters become ‘xxxxxx’, so that all units are not matched and their matching outputs are disabled again.

6.2.4 Experiment and evaluation

Since the rule data of the configurable architecture can be reconfigured as required, the propose architecture is evaluated on both FPGA and ASIC devices. The proposed architectures were designed in VHDL code and then built by Altera’s development tool Quartus II. The built results were simulated in Modelsim to verified the logic functions of the architectures. The devices selected for evaluating the proposed architectures are Altera’s products Stratix IV FPGA and HardCopy IV ASIC, where the FPGA has 424,960 ALUTs and 424,960 registers and the ASIC device has 9,774,880 HCells.

These implementations consist of 64 rule units for the purpose of verification and evaluation. Three cases, i.e. 1-character, 2-character, and 4-character rule-units, are evaluation. In all of the cases, a group unit can consist of four rule-units at most. It means that the three cases can be configured to process 1 through 4, 2 through 8, and 4 through 16 characters in parallel, respectively.

The results of implementing the configurable architectures with different data widths on FPGAs and ASICs are summarized in Table 6.3 and Table 6.4 respectively. In which, the achievable throughput is derived by multiplying the data width with the clock rate. In the evaluation on FPGAs, the hardware resource required for a character is represented by LE/char, which is calculated as follows:

$(\sharp \text{ of ALUTs} + \sharp \text{ of Registers}) * 1.25/(\text{total characters})$

Table 6.3: Implementing the configurable architecture on FPGA

Data Width/Unit	**1 Char.**	**2 Char.**	**4 Char.**
Combinational ALUTs	6,261	10,263	13,524
Dedicated logic registers	1,885	4,857	7,684
LE/Char	159	148	104
Fmax (MHz)	44.35	39.22	37.64
Data Width/Group	8-32 bits	16-64 bits	32-128 bits
Throughput (Gbps)	0.35-1.42	0.63-2.51	1.20-4.82

Table 6.4: Implementing the configurable architecture on ASIC

Data Width/Unit	**1 Char.**	**2 Char.**	**4 Char.**
Total HCells	39,649	73,000	98,232
HCell/Char	629	570	384
Fmax (MHz)	60.0	55.03	54.85
Data Width/Group	8-32 bits	16-64 bits	32-128 bits
Throughput (Gbps)	0.48-1.92	0.88-3.52	1.76-7.02

As can be seen in these tables, the resource required for a character is less when a basic rule unit can process more characters; this is might because that the resource used for matching the state pattern and the control logic is a constant overhead and is independent of the number of characters processed simultaneously in a basic rule unit. While the characters processed in parallel are more, the portion of the constant overhead in the required resource is less. As shown in the results, it is reasonable for the case that a basic rule unit processes four characters in parallel, because this case is balanced between flexibility and efficiency.

Chapter 7

Comparison of Approaches

This chapter compares the results of the proposed approaches with those of previous work. Making a fair comparison is relatively difficult, because the hardware and software approaches significantly differ in the way that they achieve the parallelism. Nevertheless, the comparison provides insight into the status of the proposed approaches.

7.1 Comparing Results

Table 7.1 summarizes the comparing results. The performance data of the other approaches are taken from the corresponding literature. For software approaches, column *Clock* is the operating clock of the CPU or GPU and column *Parallelism* is the number of processing cores. While for hardware approaches, columns *Clock* and *Parallelism* are the clock rate and the width of the data bus respectively. The throughput of a hardware implementation is generally derived by multiplying the clock rate by the data width.

In this comparison table, hardware approaches are listed in the upper rows. The approach proposed by Katashita et al. [18] is an implementation of multi-character NFAs that runs at 263 MHz and has a 512-bit data width and its throughput can achieve 134.7 Gbps. The approach of Pao and Wang [6] achieves the parallelism by using three QSV (quick sampling with on demand verification) units that runs at 230 MHz and has a 24-bit data width and consequently its throughput is 5.5 Gbps. The approach of Tripp [5] achieves the parallelism by using multiple string matching engines that can process four characters every clock and runs at 149 MHz and the achievable throughput is 4.8 Gbps. Notably, the two implementations operating at more than 200MHz clock have been accelerated by the pipeline technique.

Table 7.1: Comparison of different approaches

Approach	**Clock** (MHz)	**Parallism**	**Throughput** (Gbps)
Multi-character AC-DFA*	142	32 bits	4.5
Multi-character AC-NFA*	137	128 bits	17.6
Multi-character hybrid-FA*	66.6	64 bits	4.3
Configurable stage architecture*	123	64 bits	7.9
Configurable data-width architecture*	55	32-128 bits	1.8-7.0
multi-character NFA [18]	263	512 bits	134.7
QSV implementation [6]	230	24 bits	5.5
Parallel string matching engine [5]	149	32 bits	4.8
Multicore CPU (IBM Cell/B.E.) [19]	3,200	8 cores	40
Multicore GPU (Nvidia) [24]	1,300	30 cores	15.6
Cray XMT [25]	N/A	128 nodes	28.0
Multicore CPU (Intel Xeon) [22]	2,260	32 cores	34.0

* Proposed architectures.

The remaining rows of the table are for software approaches. The approach of Scarpazza et al. [19] was implemented on an IBM Cell/B.E. processor which has eight SPEs (synergistic processing element); the throughput of each SPE is 5 Gbps and jointly is 40 Gbps. The approach of Tumeo et al. [24] was implemented on an Nvidia GPU Tesla C1060, which works at 1296 MHz (shader clock) and has thirty cores, and the throughput can achieve 15.6 Gbps. In the later work of Tumeo et al. [25], several architectures are evaluated where this table lists only the result with the highest performance, which is evaluated on a Cray XMT with 128 processors; the resulting throughput is 28Gbps. The approach of Yang and Prasanna [22] was implemented on a 32-core Intel Manycore Testing Lab machine based on the Intel Xeon X7560 processors, which is an 8-core 'Nehalem' running at 2.26 GHz, and the resulting throughput is 34 Gbps.

The top five rows list the results of the approaches proposed in this thesis. The proposed multi-character AC-DFA and two configurable architectures are implemented in ASIC devices, and the proposed multi-character AC-NFA and hybrid AC-FA approaches are implemented in FPGA devices. In addition, the architecture with configurable stage scheme has been optimized by a two-stage pipeline circuit. The results of the architecture with configurable data-width scheme is obtained by the implementation of 4-character units.

Among the proposed multi-character string matching approaches, the AC-NFA approach has the best performance due to the simplicity in architecture, while its circuit needs to be rebuilt when the keyword set is changed and is suitable to be implemented in programmable devices, such as FPGAs. The AC-DFA approach can operate at a higher clock rate; nevertheless, the multi-character AC-DFA approach would suffer from the problem of explosive transitions as the number of characters inspected in parallel increases. Although the proposed configurable data-width architecture operates at a relative low clock-rate, it can be configured to process more characters in parallel and can obtain a reasonable performance as compared with the AC-DFA approach. In addition, techniques of pipelining can be considered to optimize the operation clock of the configurable string matching architecture.

7.2 Discussions

The advantages of the hardware string matching accelerator are revealed from this comparison. The modern CPUs are sophisticated products that can run at very high speed and have wide data width, while they are designed for general purposes. It is worth to note that a simple hardware string matching accelerator running at much lower clock can achieve the compatible throughput with respect to a software program running at a very powerful CPU. Moreover, a hardware string matching accelerator that can inspect multiple characters in parallel can achieve multiplied throughput at the same clock rate. As comparing with the software approaches that process multiple texts in multiple threads, it is more intuitive in a real application that the hardware approaches process multiple characters in parallel.

The proposed work aims to propose a systematic approach for deriving multi-character transitions and develop high efficient string matching engines capable of inspecting multiple characters in parallel, and builds the implementations mainly for verifying the effectiveness of architecture. The obtained results are preliminary and can be improved further. For instance, the priority multiplexer are used intensively in the proposed architectures and dominates the performances of the proposed architectures. The structure of priority multiplexer is much similar to the structure of CAM, which have been researched and improved in many previous works [33,34]. Therefore, it is considerable to improve the structure of priority multiplexer by referring to the structure of CAM. The performances of the proposed architecture should be improve further when the priority multiplexer is improved.

Chapter 8

Conclusions

This thesis first presents three approaches including AC-DFA, AC-NFA, and hybrid AC-FA approaches to implement the AC-algorithm. The AC-DFA approach can be implemented in a deterministic circuit while is inefficient in space. In contrast, the AC-NFA approach is efficient in space while is nondeterministic in implementation. Therefore, this thesis proposes a hybrid AC-FA that combines both the advantages of the AC-DFA and AC-NFA approaches, i.e. efficiency in space and being deterministic in implementation. The deterministic implementation is enable to design a general architecture of string matching to process various keyword sets.

Next, an intuitive algorithm is proposed to derive multi-character transitions from an AC-DFA, an AC-NFA, or a hybrid AC-FA, where each transition can match multiple characters at a time. This derivation algorithm also includes using assistant transitions and a pseudo state to resolve the alignment problem. Several architectures are also proposed to implement the derived multi-character AC-DFA, AC-NFA, and hybrid FA, respectively. Moreover, configurable architectures are also proposed to provide flexibility in applications. Evaluations are performed for the proposed architectures, respectively, to demonstrate their properties.

In summary, the proposed architectures of multi-character transition string matching engine are simple and intuitive, allowing for its easy implementation for any required number of characters to be inspected in parallel. As a result, the proposed architectures can achieve efficient performance by inspecting multiple characters in parallel, while maintain the efficiency in the hardware implementation.

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[2] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2628–2639 vol.4, 2004.

[3] Xinyan Zha and S. Sahni. Highly compressed Aho-Corasick automata for efficient intrusion detection. In *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on*, pages 298–303, 2008.

[4] Mansoor Alicherry, Muthusrinivasan Muthuprasanna, and Vijay Kumar. High speed pattern matching for network IDS/IPS. In *Network Protocols, 2006. ICNP'06. Proceedings of the 2006 14th IEEE International Conference on*, pages 187–196. IEEE, 2006.

[5] Gerald Tripp. A parallel string matching engine for use in high speed network intrusion detection systems. *Journal in Computer Virology*, 2(1):21–34, 2006.

[6] Derek Pao and Xing Wang. Multi-stride string searching for high-speed content inspection. *The Computer Journal*, 55(10):1216–1231, 2012.

[7] Vahid Rahmanzadeh and MohammadBagher Ghaznavi-Ghoushchi. A multi-Gb/s parallel string matching engine for intrusion detection systems. In *Advances in Computer Science and Engineering*, volume 6 of *Communications in Computer and Information Science*, pages 847–851. Springer Berlin Heidelberg, 2009.

[8] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[9] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. In *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, pages 44–51. IEEE, 2003.

[10] Wei Lin and Bin Liu. Pipelined parallel AC-based approach for multi-string matching. In *Parallel and Distributed Systems, 2008. ICPADS'08. 14th IEEE International Conference on*, pages 665–672. IEEE, 2008.

[11] Derek Pao, Wei Lin, and Bin Liu. A memory-efficient pipelined implementation of the Aho-Corasick string-matching algorithm. *ACM Trans. Archit. Code Optim.*, 7(2):10:1–10:27, October 2010.

[12] Nan Hua, Haoyu Song, and T. V. Lakshman. Variable-stride multi-pattern matching for scalable deep packet inspection. In *INFOCOM 2009, IEEE*, pages 415–423, 2009.

[13] V. Dimopoulos, I. Papaefstathiou, and D. Pnevmatikatos. A memory-efficient reconfigurable Aho-Corasick FSM implementation for intrusion detection systems. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pages 186–193, 2007.

[14] YE Yang, Viktor K Prasanna, and Chenqian Jiang. Head-body partitioned string matching for deep packet inspection with scalable and attack-resilient performance. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[15] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, CoNEXT '07, pages 1:1–1:12. ACM, 2007.

[16] Yutaka Sugawara, Mary Inaba, and Kei Hiraki. Over 10Gbps string matching mechanism for multi-stream packet scanning systems. In *Field Programmable Logic and Application*, volume 3203 of *Lecture Notes in Computer Science*, pages 484–493. Springer Berlin Heidelberg, 2004.

[17] N. Yamagaki, R. Sidhu, and S. Kamiya. High-speed regular expression matching engine using multi-character NFA. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 131–136, 2008.

[18] T. Katashita, A. Maeda, K. Toda, and Y. Yamaguchi. A method of generating highly efficient string matching circuit for intrusion detection. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–4, 2006.

[19] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. Exact multi-pattern string matching on the Cell/B.E. processor. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 33–42. ACM, 2008.

[20] Leena Salmela, Jorma Tarhio, and Jari Kytöjoki. Multipattern string matching with q-grams. *Journal of Experimental Algorithmics (JEA)*, 11:1–1, 2007.

[21] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. High-speed string searching against large dictionaries on the Cell/BE processor. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[22] Y-HE Yang and Viktor K Prasanna. Robust and scalable string pattern matching for deep packet inspection on multicore processors. *Parallel and Distributed Systems, IEEE Transactions on*, 24(11):2283–2292, 2013.

[23] Oreste Villa, Daniel Chavarria-Miranda, and Kristyn Maschhoff. Input-independent, scalable and fast string matching on the Cray XMT. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[24] Antonino Tumeo, Oreste Villa, and Donatella Sciuto. Efficient pattern matching on GPUs for intrusion detection systems. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 87–88. ACM, 2010.

[25] Antonino Tumeo, Oreste Villa, and Daniel G Chavarría-Miranda. Aho-Corasick string matching on shared and distributed-memory parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 23(3):436–443, 2012.

[26] D Herath, C Lakmali, and R Ragel. Accelerating string matching for bio-computing applications on multi-core CPUs. In *Industrial and Information Systems (ICIIS), 2012 7th IEEE International Conference on*, pages 1–6. IEEE, 2012.

[27] Jennifer Stephenson and Paul Metzgen. *Logic Optimization Techniques for Multiplexers*. Altera Literature, 2004.

[28] Yi-Hua E Yang, Weirong Jiang, and Viktor K Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 30–39. ACM, 2008.

[29] C.R. Clark and D.E. Schimmel. Scalable pattern matching for high speed networks. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 249–257, 2004.

[30] Reetinder Sidhu and Viktor K Prasanna. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 227–238. IEEE, 2001.

[31] Snort. https://www.snort.org.

[32] Benfano Soewito. Packet inspection on programmable hardware. *Computer Engineering and Intelligent Systems*, 4(2):57–68, 2013.

[33] Kenneth J Schultz. Content-addressable memory core cells a survey. *INTEGRATION, the VLSI journal*, 23(2):171–188, 1997.

[34] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. *Solid-State Circuits, IEEE Journal of*, 41(3):712–727, 2006.