FAST, EFFICIENT AND DYNAMICALLY OPTIMIZED DATA AND HARDWARE ARCHITECTURES FOR STRING MATCHING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SALIH ZENGIN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

SEPTEMBER 2014

Approval of the thesis:

FAST, EFFICIENT AND DYNAMICALLY OPTIMIZED DATA AND HARDWARE ARCHITECTURES FOR STRING MATCHING

submitted by **SALIH ZENGIN** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Prof. Dr. Gönül Turhan Sayan
Head of Department, **Electrical and Electronics Engineering** _____

Prof. Dr. Prof. Dr. Hasan Cengiz Güran
Supervisor, **Electrical and Electronics Engineering Dept., METU** _____

Assoc. Prof. Dr. Şenan Ece Schmidt
Co-supervisor, **Electrical and Electronics Engineering Dept., METU** _____

Examining Committee Members:

Prof. Dr. Semih Bilgen
Electrical and Electronics Engineering Dept., METU _____

Prof. Dr. Hasan Cengiz Güran
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Dept., METU _____

Assoc. Prof. Dr. Halit Oguztüzün
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Nail Akar
Electrical and Electronics Engineering Dept., Bilkent University _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: SALIH ZENGIN

Signature :

ABSTRACT

FAST, EFFICIENT AND DYNAMICALLY OPTIMIZED DATA AND HARDWARE ARCHITECTURES FOR STRING MATCHING

Zengin, Salih
Ph.D., Department of Electrical and Electronics Engineering
Supervisor : Prof. Dr. Prof. Dr. Hasan Cengiz Güran
Co-Supervisor : Assoc. Prof. Dr. Şenan Ece Schmidt

September 2014, 105 pages

Many fields of computing such as network intrusion detection employ string matching modules (SMM) that search for a given set of strings in their input. An SMM is expected to produce correct outcomes while scanning the input data at high rates. Furthermore, the string sets that are searched for are usually large and their sizes increase steadily.

In this thesis, motivated by the requirement of designing fast, accurate and efficient SMMs; we propose a number of SMM architectures that employ Bloom Filters to compactly represent the large amounts of data for the string sets. The proposed architectures address the well-known slowdown problem of the Bloom Filters because of the verifications of the positive matches.

To this end, the first contribution of the thesis is Double Bloom Filter SMM (DBF-SMM) which employs a second Bloom Filter which acts as a verification engine. We present an analysis, evaluation and implementation of the DBF-SMM. We further verify the required functionality of the DBF-SMM by modeling and testing the architecture in SystemC environment. Our analytical and implementation results demonstrate that DBF-SMM is superior to the existing Bloom Filter based SMM designs in terms of sustainability of the response time with high string storage efficiency and hardware scalability.

DBF-SMM is designed for fixed size strings. The second contribution of the thesis is a finite automaton-based design that stores variable size strings as state transitions between characters. To this end, we first identify the classes of state transitions. We then modify the implementation of the well-known Aho-Corasick algorithm to effectively store and query the appropriate transition classes in a hardware architecture that features Bloom Filters and CAM-RAM look-up tables. The Bloom Filter in this architecture is realized as a DBF-SMM. The proposed SMM achieves a memory efficiency that is superior to all previous SMM designs together with fast and scalable hardware design.

Keywords: Network-level security and protection, network intrusion detection system, deep packet inspection, content filtering, string matching, pattern matching, Bloom filter, Finite Automaton, field programmable gate array (FPGA), SNORT, Aho-Corasick

ÖZ

DİZİ EŞLEME AMAÇLI VERİMLİ VERİ VE DONANIM MİMARİLERİ

Zengin, Salih
Doktora, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Prof. Dr. Prof. Dr. Hasan Cengiz Güran
Ortak Tez Yöneticisi : Doç. Dr. Şenan Ece Schmidt

Eylül 2014 , 105 sayfa

Verilen bir dizi kümesini kendi girişinde arayan dizi eşleme modülleri (SMM), ağ sızıntı tespit sistemleri gibi bir çok hesaplama alanında kullanılır. Bir SMM, giriş verisini yüksek hızlarda tararken, doğru sonuçlar üretmesi beklenir. Ayrıca, aranan dizi kümeleri genelde büyüktür ve boyutları da sürekli artmaktadır.

Bu tezde, hızlı, doğru ve verimli tasarım gerekliliği motivasyonu ile, büyük miktarda veriyi dizi kümeleri için sıkıştırarak temsil etmek maksadıyla Bloom Filter'leri kullanan bir kaç SMM yapısı önerilmektedir. Bu yapılar, Bloom filtrelerin olumlu eşleme sonuçlarının doğrulanması sebebiyle oluşan ve iyi bilinen yavaşlama problemini çözmeyi hedefler.

Bu amaçla, tezin ilk katkısı olarak, doğrulama maksatlı olarak kullanılan ikinci bir filtre içeren Çift Bloom filtreli dizi eşleme modülü (DBF-SMM) oluşturulmuştur. DBF-SMM'in analiz, değerlendirme ve uygulamaları gösterilmiştir. Ayrıca, DBF-SMM'in istenen işlevi, SystemC ortamında ilgili yapı modellenip test edilerek doğrulanmıştır. Analiz ve uygulama sonuçlarımız, DBF-SMM'in Bloom filtre tabanlı diğer SMM tasarımlarından tepki süresinin yüksek dizi saklama verimliliği ve donanım ölçeklendirilebilirliliği ile korunması açısından daha üstün olduğunu gösterir.

DBF-SMM sabit boyutlu diziler için tasarlanmıştır. Tezin ikinci katkısı ise, karakterler arası durum geçişleri ile değişken boyutlu dizileri saklayan sonlu makine tabanlı

tasarımlardır. Bu maksatla, durum geçişleri sınıfları tanımlanmıştır. Daha sonra, iyi bilinen Aho-Corasick algoritması gerçeklenmesi, Bloom filtreleri ve CAM-RAM tablolarını kullanarak uygun geçiş sınıflarını etkili bir şekilde donanımda saklayacak ve sorgulayacak şekilde değiştirilmiştir. Bu yapıda Bloom filtre DBF-SMM olarak gerçeklenmiştir. Önerilen SMM yapısı, bütün diğer SMM tasarımlarından üstün hafıza verimliliğine, hızlı ve ölçeklenebilir donanım tasarımıyla sahiptir.

Anahtar Kelimeler: Ağ seviyesi güvenlik ve koruma, ağ saldırı tespit sistemi, derinlemesine paket denetimi, içerik filtreleme, dizi eşleme, patern eşleme, Bloom filtresi, Durum Makinesi, alanda programlanabilir kapı dizisi, SNORT, Aho-Corasick

To My Family

ACKNOWLEDGMENTS

I would like to express my special thanks to my supervisors Prof. Dr. Cengiz Hasan Güran and Assoc. Prof. Dr. Şenan Ece Schmidt. My special thanks go to TÜBİTAK SAGE. Finally, I would like to thank my family and friends who supported me throughout the whole time.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

AC	Aho-Corasick
AC-wDFA	Aho-Corasick based wDFA
AMU	Approximate Matching Unit
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
BFbAC-wDFA	Bloom Filter based Aho-Corasick wDFA
BFwDFA	Bloom Filter based w-byte (Multi-byte) Automaton
BF	Bloom Filter
B-FSM	BaRT-based Finite State Machine technology
BRAM	Block RAM
CAM	Content Addressable Memory
ClamAV	Clam AntiVirus
CLB	Configurable Logic Block
CU	Control Unit
D^2FA	Delayed Input DFA
DBF	Double Bloom Filter
DBF-SMM	Double Bloom Filter-String Matching Module
DFA	Undeterministic Finite Automaton
DMC	Dynamic Match Counter
DMR	Dynamic Match Rate
EMU	Exact Matching Unit
FastFA	Fast Finite Automaton
FPGA	Field Programmable Gate Array
FSAM	Fast Scalable Automaton Matching
GBF	Generalized Bloom Filter
Gbps	Gigabit per second
HDL	Hardware Description Languages
IEEE	Institute of Electrical and Electronics Engineer

IEEE-SA	Institute of Electrical and Electronics Engineers Standards Association
ISE	Integrated Software Environment
LUT	Look-up Table
MBFbAC-wDFA	Multi-Bloom filter based Aho-Corasick wDFA
MCU	Monitor and Control Unit
NFA	Deterministic Finite Automaton
NIDS	Network Intrusion Detection System
NMC	Normalized Memory Consumption
NMCx	Normalized Memory Complexity
OSCI	Open SystemC Initiative
RAM	Random Access Memory
RTL	Register Transfer Language
SBF-SMM	Single Bloom Filter-String Matching Module
SMM	String Matching Module
SR	State Register
TCAM	Ternary Content Addressable Memory
UTVU	Unconsecutive Transition Verification Unit
VHDL	Very High Speed Integrated Circuit HDL
VU	Verification Unit
wDFA	w-byte (Multi-byte) Automaton

CHAPTER 1

INTRODUCTION

String matching is the task of searching for a given set of strings in a given larger string. The string matching problem manifests itself in many fields of computing including text retrieval, computational biology and signal processing [27]. *Network intrusion detection systems* (NIDS) is a very prominent area that employs *string matching module* (SMM) to search for *attack strings* in the incoming packets [15, 38, 16, 21, 39, 41, 20, 36, 42, 40, 24, 26, 12].

SMMs are expected to scale with the increasing amounts of data and the processing rate requirements in computing applications. In particular, the sizes of the attack signature sets of NIDS steadily grow to include the new signatures while maintaining the signatures of old and rare attacks which can still occur and harm the system. The SMMs of NIDS which are installed at the traffic aggregation points such as routers [14] are required to scan the incoming traffic at gigabit per second (Gbps) rates. Low response times to reach such scan rates can be achieved by SMM implementations on hardware platforms which must take the hardware logic and memory resources into account [16, 19, 43].

Bloom Filters [10] are hashing data structures which fulfill these requirements and they are frequently employed for building SMMs for network applications [13, 18]. A Bloom Filter compactly stores a given string set and queries it in constant time for the input without any false negatives. However, they perform approximate string matching with a false positive probability that increases with the size of the stored string set. Works such as [15, 16, 37] employ Bloom Filters for quickly filtering out the input that does not match the string set and an additional matching engine without

any false results for verifying the positive outcomes. The verification engine is a slower component than the Bloom Filter, hence, frequent positive outcomes because of true and false positives slow down the SMM.

Motivated by the requirement of fast, accurate and efficient SMMs, in this thesis, we propose a number of Bloom Filter based SMM architectures which exploit the compact data representation and fast response features of Bloom Filters.

To this end, the first contribution of this thesis is the novel Double Bloom Filter SMM *(DBF-SMM) architecture*. DBF-SMM has two Bloom Filters that concurrently query the incoming strings. The first Bloom Filter stores the entire string set and its no-match results can be trusted thanks to the zero false negatives. The second Bloom filter stores a limited set of strings so that its false positive probability is almost zero and its match results do not need verification. Hence, the average response time of the SMM decreases with the ratio of the matches detected by the second Bloom Filter. The content of the second Bloom Filter is dynamically adapted to the current input data to increase the detected matches while maintaining the almost zero false positive probability. In the best case, all of the positive matches for the incoming strings are detected by the second Bloom Filter and the response time of the DBF-SMM is equal to the response time of the Bloom Filters without any verification. We propose DBF-SMM as a fast SMM architecture to be implemented in hardware. Accordingly we take logic and memory resources in consideration.

DBF-SMM is a fixed-size string matching SMM. Matching variable size strings require keeping track of the order of string pieces with a deterministic finite automaton. *The second contribution of the thesis is the design of a novel Bloom Filter based Multi-byte Deterministic Finite Automaton (BFwDFA) which employs DBF-SMM as the Bloom Filter.* To this end, we propose a modified version of Aho-Corasick (AC) string matching algorithm in order to map a variable size string set into a BFwDFA. Different than previous automaton based implementations, our architecture eliminates the need to completely store all of the state transition rules from any state to some states that are zero or one transition away from the initial state, while consuming multiple bytes at a time with limited memory consumption. In this variable size SMM architecture, state transition rules and their associated transition conditions (ex-

cept eliminated transitions) are stored in a verification unit and in a BF, respectively. This enables us to achieve high memory efficiency that increases with the number of transitions eliminated. Besides, high rate of *state transitions per time* can be achieved in the case of executing high number of eliminated transitions, where BF is not expected to frequently output positive results. DBF-SMM enhances this variable size SMM as the BF to store the appropriate frequently occurring state transitions and a look-up table consisting of CAM and RAM to store the rest of the frequent transition rules.

The outline of the work in this thesis study to realize these two main contributions is as follows:

- The design, analysis and evaluation of the DBF-SMM according to the response time, correctness and the hardware resource requirement metrics
- Extension of DBF-SMM to a parallel implementation together with its analysis and evaluation
- Detailed description of the selection of the design parameters of the DBF-SMM
- One possible method of dynamically updating the content of the second Bloom Filter according to the current input
- Numeric evaluation results comparing DBF-SMM to the standard Bloom Filter based SMM under different design parameter values and FPGA implementation results of DBF-SMM that demonstrate the improvement in the response time and the implementation scalability
- Construction of SystemC model of DBF-SMM and verifying the required functionality of the proposed architecture
- Formal definition of *Multi-byte Deterministic Finite Automaton (wDFA)*, whose input symbols consist of one or more ASCII characters. Classification of the states, state transition rules and state transition conditions according to their shortest distance from the initial state. Basic hardware implementations of wDFA with RAM and/or CAM to constitute a basis for comparison.

- Bloom Filter based wDFA (BFwDFA) which eliminates storing all of the transitions from any state to the initial state (transitions to the initial state). In this architecture, state transition rules and their associated transition conditions (except transitions to the initial state) are stored in a verification unit and in a BF, respectively. This architecture enables us to achieve high memory efficiency that increases with the number of transitions eliminated. Besides, high rate of *state transitions per time* can be achieved in the case of high number of transitions to the initial state, where BF is not expected to frequently output positive results.

- A slight modification of Aho-Corasick (AC) string matching algorithm in order to store a given string set in a wDFA which yields an Aho Corasick-based automaton (AC-wDFA).

- A preprocessing C++ program to store any given string set by the modified AC algorithm on a wDFA. We demonstrate the significant features of AC-wDFA and calculate the memory usage efficiency of the proposed variable string size matching engine by storing the well-known SNORT string set in an AC-wDFA.

- Bloom Filter based implementation of AC-wDFA (BFbAC-wDFA) in order to achieve higher memory efficiency than AC-wDFA. This is achieved by eliminating the transitions from any state to some states that are one transition away from the initial state (depth-1 transitions). Similar to BFwDFA, transitions to the initial state are also inferred instead of completely storing them in a look-up table. Actually, depth-1 transitions are inferred by a CAM and a RAM storing only the associated input symbols and the next states respectively. The novel advantage of this automaton is limiting the required memory to infer the depth-1 transitions when the machine is needed to consume multiple-bytes at a time.

- Employing DBF-SMM in BFbAC-wDFA, namely, Multi-Bloom Filter based AC-wDFA (MBFbAC-wDFA). BF-based architectures need verification units that are typically slow engines due to eliminate the false positive outcomes, which slows down the execution of the machine. To overcome this side effect, we previously proposed double Bloom Filter structure (DBF-SMM). In this architecture, frequent consecutive transition rules are approximately stored in

a BF where frequent unconsecutive transition rules are stored completely in a look-up table consisting of a CAM and a RAM. The overall machine can conserve the state transition rate while consuming low memory.

- Evaluation of the proposed architectures, wDFA, BFwDFA, BFbAC-wDFA and MBFbAC-wDFA with signature set of SNORT v2.9 and our results are also compared with other related works. To achieve fair comparison, the hardware complexity of the used memories are also considered.

The remainder of the thesis is organized as follows. First of all, Chapter 2 formalizes the string matching problem with its performance evaluation metrics. Then, Bloom Filters are introduced and their string matching applications in the literature are given. Chapter 3 presents design, analysis and evaluation of the novel DBF-SMM by comparing it with the single-BF version and the architecture is extended to the parallel implementation including its analysis and evaluation in Chapter 4. Chapter 5 exemplifies dynamically updating the content of the second Bloom Filter according to the current input. Chapter 6 demonstrates a number of different design cases for both of the single and parallel implementations of the DBF-SMM and depicts the evaluation results. These cases are also implemented on an FPGA and the corresponding resource consumptions and clock periods are reported by Chapter 7. In Chapter 8 SystemC model of the DBF-SMM and the verification of the architecture are explained. Chapter 9 is dedicated to automaton-based variable size string matching architectures. In this chapter, firstly, Multi-byte Deterministic Finite Automaton is defined formally, and then its BF-based version is described. After that, Aho-Corasick string matching automaton is summarized with its multi-byte type. At the end of the chapter, SNORT string set is mapped into an automaton emphasizing some significant observations. Chapter 10 applies the idea behind DBF-SMM into Aho-Corasick and BF based automatons that are applied as variable size string matching machines. Evaluation of the proposed architectures and the comparison of the related works are also presented in this chapter. Lastly, Chapter 11 summarizes the most significant outcomes of this study and some possible advancements as future work.

CHAPTER 2

STRING MATCHING WITH BLOOM FILTERS

2.1 String Matching: Problem Formulation and Performance Metrics

Let the string database is represented as a finite set of strings $\mathcal{S} = \{S_1, S_2, \cdots, S_n\}$. For the scope of this thesis, we assume that the strings consist of fixed number of characters where each character is represented by a single byte similar to an ASCII character. We remove this constraint by extending our work to include variable size strings starting with Chapter 9. The number of strings in $\mathcal{S}$ is $n = |\mathcal{S}|$. For a string S_j with a length of w, *string matching* determines if $S_j \in \mathcal{S}$ is true.

String matching is realized with a String Matching Module (SMM) which stores $\mathcal{S}$ and scans an input stream of bytes P by shifting a search window of w bytes along P. The string representations in libraries such as SNORT [5] determine the string boundaries in bytes. Hence, for the rest of this thesis we assume that the search window is shifted byte-by-byte. SMM queries the string S_j that is enclosed in the current search window P_i^w and produces an outcome which is a *match* or a *no-match*.

Next, we define the metrics to evaluate a given SMM that is operating as described above.

Correctness: We call the events $S_j \in \mathcal{S}$ and $S_j \notin \mathcal{S}$ *positive* and *negative* respectively for the rest of the thesis. The probability that a given w-byte string S_j is positive is $P_{\mathcal{S}}$. We assume that this probability is the same for all w-byte strings that are queried.

Accordingly, we define the following probabilities for the SMM outcomes per query;

$$
\begin{aligned}
&\text{True positive probability: } Ptp &&= P(\text{match}|\text{positive}),\\
&\text{False positive probability: } Pfp &&= P(\text{match}|\text{negative}),\\
&\text{True negative probability: } Ptn &&= P(\text{no-match}|\text{negative}),\\
&\text{False negative probability: } Pfn &&= P(\text{no-match}|\text{positive}).
\end{aligned}
\tag{2.1}
$$

For a given SMM design, it is desired to have high *precision* and *accuracy* as defined in Section 3.3.2 which requires that Pfn and Pfp are low.

Response Time and Scan Rate: We define the response time T as the time to query a given string in the $\mathcal{S}$ for a possible match. The SMM executes $1/T$ queries per unit time. If the SMM shifts the search window b bits for each query on the input P, the scan rate is $R = b/T$ bits/sec.

Complexity and Resource Requirements: The amount of required resources to store the $\mathcal{S}$ together with the time complexity of querying and adding a new string to $\mathcal{S}$ determine the feasibility and the scalability of a given SMM design.

Variable Size String Support: Variable size string support is defined as an ability of storing a set of strings $\mathcal{S} = \{S_1, S_2, \cdots, S_n\}$, where each string consists of a different number of characters.

2.2 Bloom Filters

Bloom Filters are memory-efficient, probabilistic, multi-hashing data structures with many applications in computer networks [10, 13, 18]. The focus of our thesis is string matching, hence we refer to the data that is stored in the Bloom Filters as strings.

A Bloom Filter consists of a linear vector v with m bits and k independent hash functions, $\mathcal{H} = \{h_1, h_2, \ldots, h_k\}$ as illustrated in Fig.2.1.

Storing $\mathcal{S}$ in a Bloom Filter starts with clearing all m bits in v. Then, for each element $S_j \in \mathcal{S}$, each hash function $h_i(S_j)$ maps S_j to v by setting the corresponding bits to logic 1 for $i = 1 \cdots k$, and $j = 1 \cdots n$.

Querying an input string S_j computes all $h_i(S_j)$ to find the respective bit positions in

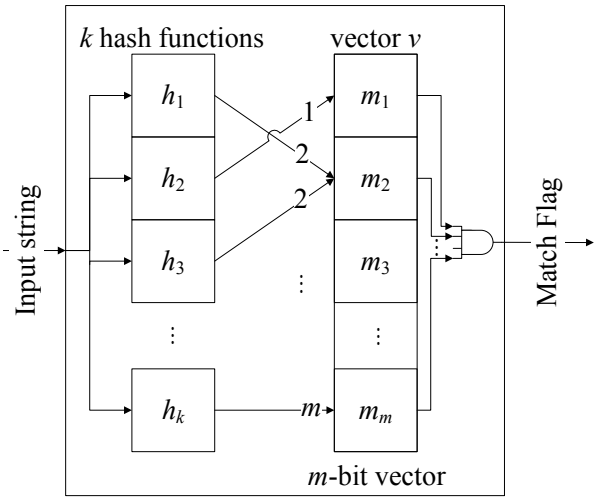


Figure 2.1: Bloom Filter architecture. All of the hash values for an input string are mapped to the m-bit positions in the vector v.

v and returns a *match* result if all of these bits are set to logic 1. The time needed to store and query an element only depends on the number of serially implemented hash functions. These hash functions can be executed concurrently in hardware resulting in $O(1)$ time complexity. This property not only supports the scalability of the querying with respect to $|\mathcal{S}|$ but also it enables the easy storing of a new element. Deleting an element from a Bloom Filter is not a straight forward operation and requires a more complicated design such as counting Bloom Filters [18].

A Bloom Filter always returns a match result if the corresponding string is stored in v. However, it can generate false positive results since all of the ones checked during the query are not necessarily set by a single element in the set $\mathcal{S}$. Furthermore, if the hash functions are not one-to-one, it is possible that they map a queried string that is not in $\mathcal{S}$ to all 1 locations in v. Fig. 2.2 shows these two examples for the false positives.

The analysis for Bloom Filters in the literature [18, 13] presents the following results.

Consider a Bloom Filter with k hash functions and the vector v with a size of m to store $\mathcal{S}$. Assume that the outputs of the hash functions are uniformly and randomly distributed to the range of $\{1, 2, \cdots, m\}$. Then, the false positive probability Pfp and the corresponding optimal number of hash functions, k_{opt}, that minimize Pfp

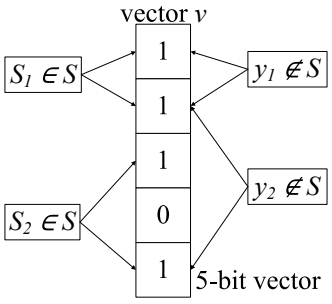


Figure 2.2: S_1 and S_2 are members of the set $\mathcal{S}$ and they are approximately represented by the Bloom Filter with $k = 2$ and $m = 5$ bits. The queried strings $y_1, y_2 \notin S$ falsely produce match outcomes because the m-bit vector positions pointed by the calculated hash values are set to logic 1 by S_1 and S_2.

are;

$$Pfp = (1 - e^{-k \cdot |\mathcal{S}|/m})^k, \tag{2.2}$$

$$k_{opt} = \frac{m_{opt}}{|\mathcal{S}|} \cdot ln2. \tag{2.3}$$

By replacing k in Equation (2.2) as k_{opt} in Equation (2.3);

$$k_{opt} = -log_2(Pfp). \tag{2.4}$$

Accordingly, the Bloom Filter's worst-case space complexity is $O(|\mathcal{S}|)$ under the constant false positive rate. Note that the time and space complexities are independent of the size of the strings stored in a Bloom Filter. However, the hardware complexity of the hash functions is dependent on the string size as given in Section 3.6.

For a given k and Pfp one can derive the ratio between $|\mathcal{S}|$ and m where m is not necessarily optimal as follows:

$$\frac{|\mathcal{S}|}{m} = -\frac{\ln(1 - e^{\frac{\ln(Pfp)}{k}})}{k}. \tag{2.5}$$

2.3 Approximate String Matching with Bloom Filters

SMMs featuring Bloom Filters are prominently employed in computer network security applications [18] such as intrusion detection by deep packet inspection. To this

end, an SMM scans each packet P to check whether it contains any known attack signature string. Hence the *attack signatures* constitute the string set $\mathcal{S}$. A match results in actions that include marking or dropping the packet.

A Bloom Filter does not have any false negatives (misses). Therefore, it can be employed in order to quickly filter out the input strings that do not match the string database [15, 16, 37]. However, the positive query results must be verified by another engine without any false positives. Most of the time this is a relatively slow exact matching engine. If the match outcomes are frequent, then the subsequent verifications result in a significant decrease of the scan rate [30].

[15] proposes a set of parallel Bloom Filters, each of which approximately represents only a set of fixed size strings. Therefore, variable size strings can be represented with overall architecture. However, the proposed machine is implemented as a prototype system that stores only 32-byte fixed-size strings. On one hand, the no-match outcomes of each Bloom filter can be trusted because of the no false negatives of the Bloom Filters. On the other hand, any match signal from the filters is verified by a slow software component to check against the false positive probability, therefore, the engine slows down to the speed of the verification engine under frequent matches. In [16] variable size strings are stored in a finite state machine by splitting long strings into multiple short length substrings with an Aho-Corasick based scheme. In this architecture, state transition conditions are stored into a set of Bloom Filters. If any BF match occurs, then slow off-chip memory is accessed to get the next state information, which slows down the machine in the case of high BF matches. Multiple SMMs are also employed to gain higher scan rates.

A sub-linear time (BFAST) algorithm is proposed to accelerate the Bloom Filter operation for pre-filtering of the clear packets in [25]. In this algorithm, the shift distance of the search window is determined by a heuristic similar to the bad-character heuristic used in the Boyer-Moore algorithm [11]. The results of the queries from the parallel Bloom Filters are considered as algorithmic heuristics. The false positives of Bloom Filters decrease the shift distance for the search window down to one byte at the worst case. If all Bloom Filters return match results, a sequential slow verification must be performed. A linear worst-case time option is also proposed for BFAST but

it is not implemented.

In [37], a fast scalable automaton-matching (FSAM) hardware combining pre-hashing and root-indexing techniques are proposed in order to speed up the bitmap AC-based matching process. The root-index approach compares multiple bytes in one single matching process when the machine is in the root state. Before AC matching, pre-hashing tests the input substring for the non-root states using hash functions. The slow automaton matching is skipped if the initial hashing does not produce a match for the input mapped on the current search window. Under frequent matches, slow automaton matching cannot be skipped most of the time. Root-indexing only advances the matching by a limited number of bytes k_{root} for each matching string. Consequently, if the match rate is high and the length of the matching strings is longer than k_{root}, then the advantage of the root-indexing decreases. The memory efficiency of the proposed machine is significantly low with respect to mainly Bloom Filter-based architectures because the systems is based on automaton. In addition, in order to avoid the large size in building root-index data, k_{root} is selected to be less than 2 bytes for large string sets.

Bloom filters are also employed to transmit large sets in a compact form to exchange information. In these kinds of applications, the content of vector v is transmitted among distributed network systems. However, an attacker can easily hack the system by broadcasting v vectors whose all bits are set to one, where the false positive probability becomes one. Hence, the standard Bloom filters are not secure in transferring information over untrusted medium. In [23], Generalized Bloom Filter (GBF) is proposed to prevent these types of attacks. In the architecture, an additional set of hash functions is employed to "reset" the m-bit locations during the storing process. This process results in a non-zero false negative probability at each query and limits the false positive probability and broadcasting all-one v vectors of GBF is meaningless. As a result, the GBF can provide more secure data representation with the limited non-zero error rates.

We further discuss a number of previous works in the literature in Section 10.3 after we introduce our proposed variable size SMM, which is based on automaton in 10.

CHAPTER 3

STRING MATCHING MODULE ARCHITECTURE WITH DOUBLE BLOOM FILTERS

3.1 Analysis of Fixed-size String Matching Module with a Single Bloom Filter

The motivation for our work in this thesis is the increased response time of the Bloom-Filter based SMMs because of the slow verification such as [15, 16, 37]. While the analysis presented in this Section is for fixed size strings we emphasize that our approach throughout this thesis targets decreasing the slow verification time for both fixed and variable size string matching modules.

To this end, we first present a model for such SMMs and analyze it in this section to demonstrate the slow down in the response time under frequent matches.

We call the SMM that is proposed in [15, 16, 37] a Single Bloom Filter SMM (SBF-SMM). To this end, an SBF-SMM is constructed with a fast Approximate Matching Unit (AMU) that features a Bloom Filter F_A and a slow Exact Matching Unit (EMU) that is a low cost engine without any particular optimization for a high scan rate. The response time for a query in AMU and EMU are t_{AMU} and t_{EMU} respectively where $t_{EMU} = E \cdot t_{AMU}$ and $E > 1$.

Let F_A store the string database $\mathcal{S}$ where all strings have the same fixed length of w bytes. The false positive probability of F_A that is computed by Equation (2.2) is Pfp_A.

For a queried string S_j, let Pm_A denote the probability of a match result of F_A. Then,

$$Pm_A \ = P_{\mathcal{S}} + (1 - P_{\mathcal{S}}) \cdot Pfp_A. \tag{3.1}$$

Let $T_{SBF-SMM}$ denote the average response time of the SBF-SMM per query. It is desired that $T_{SBF-SMM}$ is close to t_{AMU} to take the most advantage of the Bloom Filter component. To this end, we express $T_{SBF-SMM}$ in terms of t_{AMU} as follows:

$$\begin{aligned} T_{SBF-SMM} &= t_{AMU} + Pm_A \cdot E \cdot t_{AMU}, \\ &= t_{AMU} \cdot [1 + E(\cdot P_{\mathcal{S}} + (1 - P_{\mathcal{S}}) \cdot Pfp_A)]. \end{aligned} \tag{3.2}$$

The verifications of the match results of F_A that are either for positive events or false positives increase $T_{SBF-SMM}$. Pfp_A can be decreased by increasing the number of hash functions. However, $P_{\mathcal{S}}$ only depends on the incoming strings. $T_{SBF-SMM}$ grows linearly with $P_{\mathcal{S}}$ and approaches to $t_{AMU} + t_{EMU}$ under high $P_{\mathcal{S}}$, diminishing the advantage of the Bloom Filter.

3.2 Fixed-size String Matching Module with Double Bloom Filter: DBF-SMM

We propose a String Matching Module that features an AMU that is constructed with two Bloom Filters to mitigate the increase of the response time as $P_{\mathcal{S}}$ increases. Hence, we call this architecture a Double Bloom Filter SMM (DBF-SMM) as shown in Fig. 3.1.

The AMU consists of two Bloom Filters F_A and F_F. F_F and F_A have two independent bit vectors v_F and v_A with respective numbers of bits m_F and m_A. F_A is identical to the F_A in the SBF-SMM with a false positive probability Pfp_A. F_F stores a set of strings $\mathcal{F} \subset \mathcal{S}$. The false positive probability of F_F is purposely selected as $Pfp_F \approx 0$ by selecting $|\mathcal{F}|$ sufficiently small. Then, if F_F matches, it is far more likely that $S_j \in \mathcal{F}$ and therefore F_A also matches. Similarly, if F_A does not match, then it is exactly true that $S_j \notin \mathcal{S}$, and due to $\mathcal{F} \subset \mathcal{S}$, F_F is not expected to match.

DBF-SMM queries the string S_j that is enclosed in the current search window as defined in Section 2.2 and shifts the window byte by byte to scan the input stream P. We propose the DBF-SMM for fast SMMs that are implemented in hardware. Hence, without loss of generality, we assume that both F_F and F_A share a set of k

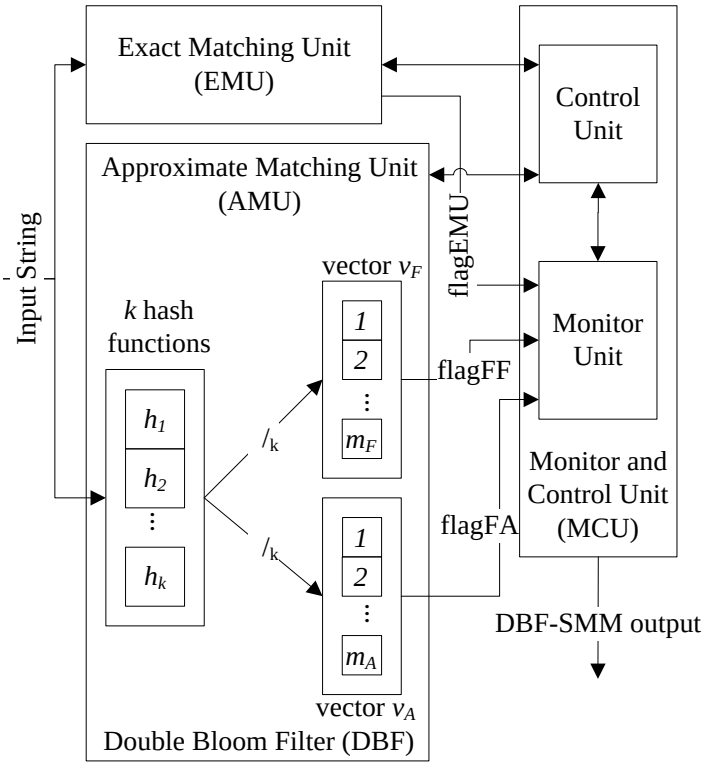


Figure 3.1: DBF-SMM Architecture.

hash functions $\mathcal{H}$ to save the expensive hardware resources. Instead, independent hash functions can be employed for each filter, resulting two times more complexity in hardware and lower scan rates. When a string S_j is to be queried, it is first concurrently mapped to v_F and v_A through the same hash function set $\mathcal{H}$ with a response time of t_{AMU}. Similar to the SBF-SMM, the DBF-SMM has an EMU with a response time of $t_{EMU} = E \cdot t_{AMU}$ and $E > 1$.

If $S_j \notin \mathcal{S}$, then F_A produces a no-match result that is correct because of the zero false negatives of the Bloom Filters. If $S_j \in \mathcal{F}$, then F_F produces a match result that does not need verification because $Pfp_F \approx 0$. In both cases, the query is completed in t_{AMU} with a no-match and a match result respectively.

For a given input S_j, a match in F_A without a match in F_F can happen either because $S_j \in \mathcal{S}$ and $S_j \notin \mathcal{F}$ or there is a false positive result with probability Pfp_A. Hence, a match of F_A without a match in F_F requires the subsequent verification by the EMU.

The Monitor and Control Unit (MCU) evaluates the match results of F_F and F_A and

either produces the query result or invokes the EMU.

3.3 Analytical Model of DBF-SMM

We first introduce the relevant probabilities for DBF-SMM and then we analyze its performance according to the metrics that we define in Section 2.1.

Let Pm_A and Pm_F denote the match probabilities per query for F_A and F_F respectively. Ptp_A and Ptp_F are the respective true positive probabilities of F_A and F_F where true positive is as defined in Section 2.1.

F_A stores the entire $\mathcal{S}$ and produces a match result for all $S_j \in \mathcal{S}$. Hence, the true positive probability of F_A is;

$$Ptp_A = P(F_A \text{ matches } |\text{positive}) = 1. \tag{3.3}$$

F_F stores $\mathcal{F} \subset \mathcal{S}$ hence, for a given input string $S_j \in \mathcal{S}$, it is possible that $S_j \notin \mathcal{F}$. Then the true positive probability of F_F is;

$$Ptp_F = P(F_F \text{ matches } |\text{positive}) = \alpha \leq 1. \tag{3.4}$$

Here, we define α, as a parameter which depends on the choice of $\mathcal{F}$.

Then the match probabilities of F_A and F_F are as follows:

$$\begin{aligned} Pm_A &= P(F_A \text{ matches } |\text{positive}) \cdot P_{\mathcal{S}} + P(F_A \text{ matches } |\text{negative}) \cdot (1 - P_{\mathcal{S}}), \\ &= P_{\mathcal{S}} + (1 - P_{\mathcal{S}}) \cdot Pfp_A. \end{aligned} \tag{3.5}$$

$$\begin{aligned} Pm_F &= P(F_F \text{ matches } |\text{positive}) \cdot P_{\mathcal{S}} + P(F_F \text{ matches } |\text{negative}) \cdot (1 - P_{\mathcal{S}}), \\ &= \alpha \cdot P_{\mathcal{S}}. \end{aligned} \tag{3.6}$$

The four combinations of all possible filter matches and their respective probabilities are enumerated below:

1. F_A does not match, F_F does not match: The probability is $P_{\bar{A}\bar{F}}$.
2. F_A does not match, F_F matches: The probability is $P_{\bar{A}F} = 0$.
3. F_A matches, F_F does not match: The probability is $P_{A\bar{F}}$.
4. F_A matches, F_F matches: The probability is P_{AF}.

We can define these probabilities in terms of match probabilities that are introduced above as follows:

$$\begin{aligned} P_{A\bar{F}} + P_{AF} &= Pm_A = P_{\mathcal{S}} + (1 - P_{\mathcal{S}}) \cdot Pfp_A, \\ P_{\bar{A}F} + P_{AF} &= P_{AF} = Pm_F = \alpha \cdot P_{\mathcal{S}}. \end{aligned} \tag{3.7}$$

Note that $P_{A\bar{F}} + P_{AF} + P_{\bar{A}\bar{F}} = 1$. Hence;

$$P_{A\bar{F}} = P_{\mathcal{S}} \cdot (1 - \alpha) + (1 - P_{\mathcal{S}}) \cdot Pfp_A \tag{3.8}$$

and,

$$\text{when there are no matches: } P_{A\bar{F}}^{P_{\mathcal{S}} \approx 0} = Pfp_A, \tag{3.9}$$

$$\text{under very frequent matches: } P_{A\bar{F}}^{P_{\mathcal{S}} \approx 1} = (1 - \alpha). \tag{3.10}$$

Note that, in the case of having independent hash functions for each filter, $P_{\bar{A}F}$ is not zero and we have an opportunity to detect some of the false positives of F_F. In other words, for a given input string S_j, if F_A does not match and F_F matches, it means that $S_j \notin \mathcal{S}$ and F_F matches falsely.

3.3.1 Response Time

The response time of DBF-SMM for querying input string S_j changes according to the outcomes of the Bloom Filters. The possible query outcomes and their corresponding response times are enumerated below:

1. Neither F_A nor F_F matches with probability of $P_{\bar{A}\bar{F}}$. No further verification or processing is required and the query finishes in t_{AMU}.
2. Both F_A and F_F match with probability of P_{AF}. This result also does not require further processing as $Pfp_F \approx 0$ and the query finishes in t_{AMU}.
3. F_A matches but F_F does not match with a probability of $P_{A\bar{F}}$. Hence, EMU verification has to follow the operation of AMU and the query finishes in $t_{AMU} + t_{EMU}$. We can look at this case in two components:
   (a) If $S_j \in \mathcal{S}$, the match is missed by F_F with probability of $P_{\mathcal{S}} \cdot (1 - \alpha)$.
   (b) If $S_j \notin \mathcal{S}$, then F_A has a false positive match with probability $(1 - P_{\mathcal{S}}) \cdot Pfp_A$.

Then, the average response time of the DBF-SMM is:

$$\begin{aligned} T_{DBF-SMM} &= t_{AMU} + E \cdot t_{AMU} \cdot P_{A\bar{F}} \\ &= t_{AMU} \cdot [1 + E \cdot (P_{\mathcal{S}} \cdot (1 - \alpha) + (1 - P_{\mathcal{S}}) \cdot Pfp_A)]. \end{aligned} \tag{3.11}$$

3.3.2 Correctness

The false positive probability of DBF-SMM is $Pfp = Pfp_F \approx 0$ because of the following:

1. If only F_A returns a match, the result is verified by EMU. Hence, no false positive occurs.
2. F_F returns a false positive match with $Pfp_F \approx 0$.

There are no false negatives because the AMU pre-screens the incoming strings without any misses.

Consequently, both the precision and accuracy of the DBF-SMM are one as shown in Equations (3.12) and (3.13).

$$Precision = \frac{Ptp}{Ptp + \underbrace{Pf_p}_{\approx 0}} \approx 1 \tag{3.12}$$

$$Accuracy = \frac{Ptp + Ptn}{Ptp + Ptn + \underbrace{Pfp + Pfn}_{\approx 0}} \approx 1 \tag{3.13}$$

3.4 DBF-SMM and SBF-SMM Response Time Comparison

Here, we compare the response times of a DBF-SMM and an SBF-SMM with identical design parameters to quantify the advantage of the DBF-SMM. Accordingly, t_{AMU}, t_{EMU} and Pfp_A are the same.

As seen in Equations (3.2) and (3.11) both $T_{SBF-SMM}$ and $T_{DBF-SMM}$ grow linearly as $P_{\mathcal{S}}$ increases. However, DBF-SMM slows down this growth with α. Hence, the difference in the response times of SBF-SMM and DBF-SMM under the same $P_{\mathcal{S}}$ is $\alpha \cdot P_{\mathcal{S}} \cdot E \cdot t_{AMU}$. If $\alpha = 0$, $T^{\alpha=0}_{DBF-SMM} = T_{SBF-SMM}$.

We propose DBF-SMM to mitigate the increase of the SBF-SMM response time as $P_{\mathcal{S}}$ increases. The respective average response times under frequent positive events are as follows:

$$T^{P_{\mathcal{S}} \approx 1}_{SBF-SMM} = t_{AMU} \cdot [1 + E], \tag{3.14}$$

$$T^{P_{\mathcal{S}} \approx 1}_{DBF-SMM} = t_{AMU} \cdot [1 + E \cdot (1 - \alpha)]. \tag{3.15}$$

It is interesting to note that when $P_{\mathcal{S}} \approx 1$, the effect of false positives are diminished. Hence, when $\alpha = 1$, DBF-SMM reaches a response time of t_{AMU}. We discuss the possible selection strategies for $\mathcal{F}$ in Section 5 such that α is maximized.

Under infrequent positive events, the EMU verifications can occur due to non-zero false positive probability. Accordingly by (3.2) and (3.11);

$$T_{DBF-SMM}^{P_{\mathcal{S}} \approx 0} = T_{SBF-SMM}^{P_{\mathcal{S}} \approx 0} = t_{AMU} \cdot (1 + E \cdot Pfp_A). \tag{3.16}$$

3.5 Selecting Design Parameters

We assume that $|\mathcal{S}|$ and E are known before when a DBF-SMM is to be designed. Accordingly, we determine k, m_A, m_F and $|\mathcal{F}|$ which are the design parameters that affect both the performance and the resource consumption of the DBF-SMM. Logic resources are required for implementing the k hash functions and memory resources are required for the bit vectors v_A and v_F.

The first parameter to be decided is k which affects Pfp_A. We limit Pfp_A by Pfp_{A_max} such that the penalty of false positives is limited to the 1% of the desired response time t_{AMU} when $P_{\mathcal{S}} \approx 0$ as in Equation (3.16). To this end, we define $Pfp_{A\_max} = 0.01/E$. Next, we determine the optimal k for Pfp_{A_max} by Equation (2.4) and optimal m_A according to (2.3) respectively.

The hash functions are shared between F_A and F_F. Once k is selected according to Pfp_{A_max}, $|\mathcal{F}|$ and m_F are determined such that $Pfp_F \approx 0$. To this end, first we need to define what is approximately 0 for Pfp_F.

The high-speed SMMs are mostly used in computer networking security applications where a match indicates that a packet is carrying a malicious string. The consequent action is usually dropping the respective packet. The only possibility for false positive matches in DBF-SMM that can lead to a packet drop by mistake is because of the non zero $Pfp_F \approx 0$. Even if there is no security application, packets can be dropped because of the buffer restrictions of the routers. Hence, we choose Pfp_F such that the probability of a packet drop after a rare false positive result is less than the existing packet drop probability in the routers. For a lightly congested core-router with a buffer of 100 packets, the packet loss probability is 10^{-11} [22]. Assuming an average payload length of 500 bytes and the SMM queries the input by 1 byte shifts, the maximum false positive probability *per query* is $Pfp_{F\_max} = 10^{-11}/500 = 2 \cdot 10^{-14}$.

Given Pfp_{F_max} and k, after determining $\frac{|\mathcal{F}|}{m_F}$ by Equation (2.5), $|\mathcal{F}|$ is selected ac-

cording to the available memory resources for m_F.

3.6 Complexity and Resource Requirements

The number of hash functions together with the length of the strings to be stored determines the logic resource consumption when implementing a given Bloom Filter on hardware. The strings that are stored in the DBF-SMM are of fixed size w bytes according to our assumption in Section 2.1.

Let $Z = w \cdot 8$ and Y denote the sizes of the input and output of a hash function in bits respectively. Each hash function maps the input $z_1, z_2, \ldots, z_Z$ to the outputs $y_1, y_2, \ldots, y_Y$ by bit-wise and (AND) and exclusive or (XOR) operations [32]. Accordingly, the hash function is defined by a matrix of bits $[a_{y,z}]$ where:
$y_i = (z_1 \text{ AND } a_{i,1}) \text{ XOR } (z_2 \text{ AND } a_{i,2}) \text{ XOR } \ldots (z_Z \text{ AND } a_{i,Z})$.

As shown in the expression above, each output bit y_i is produced by Z AND gates and an XOR gate with Z inputs. Hence the complexity of a hash function is determined by the number of outputs and inputs $O(Y \cdot Z)$. Consequently, the respective logic complexity of the hash function implementation for the DBF-SMM is $O(k \cdot Y \cdot w)$.

A hash function with Y outputs can address a memory range of 2^Y bits. Consider a vector of m bits. If $m < 2^Y$, it is possible to use additional logic to map the larger hash function output range to m. If $m > 2^Y$, m can be partitioned into blocks of 2^Y bits and perform AND on the match flags of the individual blocks.

The memory requirement is determined directly by the size of the output vectors for F_A and F_F with sizes m_A and m_F bits respectively. Hence the total memory requirement is $O(m_A + m_F)$.

Storing a string in a Bloom Filter is a constant time operation. Furthermore, provided that the EMU verifications are infrequent by maintaining $\alpha \approx 1$, querying of the strings is only carried out by AMU with a constant time complexity.

One should note that the response time and the correctness of the DBF-SMM im-

proves with increasing m_A, m_F and k. The optimal hardware design that takes the transistor counts for the hash functions and the output vectors together with the false positive probabilities is out of the scope of this thesis.

CHAPTER 4

PARALLEL IMPLEMENTATION OF DBF-SMM

The AMU in the proposed DBF-SMM architecture consists of two Bloom Filters F_A and F_F. It is possible to decrease the response time of the DBF-SMM by employing s identical AMUs that store identical contents and work in parallel as depicted in Fig.4.1. We do not aim to gain performance by EMU enhancement; hence, we assume that there is a single EMU in the system and all required verifications are performed sequentially.

Each AMU$_i$, $i = 0 \dots s-1$ checks a search window of w bytes. The search windows of the consecutive AMU's are one byte shifted with respect to each other to cover all of the substrings. To this end, the search window of AMU$_i$ receives the bytes from b_{j+i} to $b_{w+j+i-1}$ of the incoming string. Here $j = 1, 2 \dots P_{size} - w - s + 2$ denotes the start of the search window for an input byte stream P of size P_{size}. The parallel system of AMU$_0$ ...AMU$_{s-1}$ queries a total of $w + s - 1$ bytes starting from b_j to $b_{w+j+s-2}$ within t_{AMU} and shifts s bytes after each query. It is important to note that both single AMU and parallel AMU implementations execute a total of P queries to scan a P byte input. However, the parallel implementation executes s queries concurrently speeding up the scan.

When any AMU has an outcome with a match result from its F_A and a no-match from its F_F, this match must be sequentially verified by the EMU. The same four combinations of matches of F_A and F_F that are enumerated in Section 3.3.1 apply for each AMU$_i$.

Each AMU$_i$ is exposed to P_S as all bytes of P has the same P_S according to our

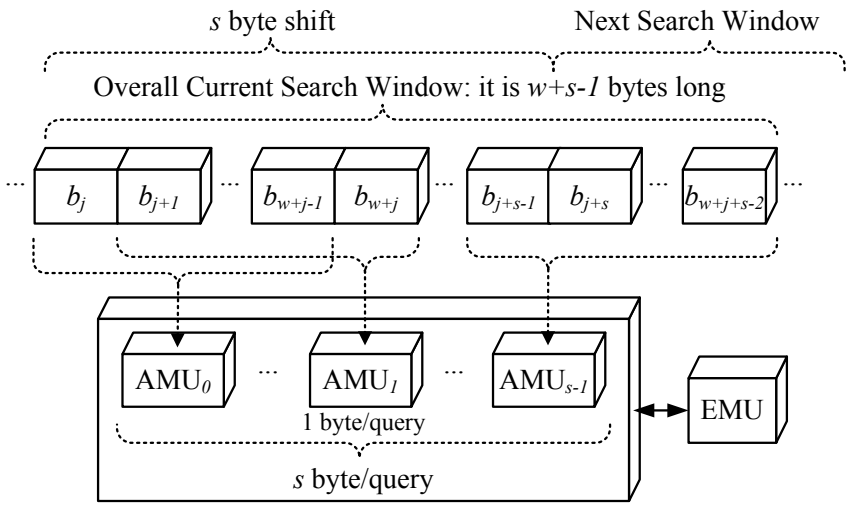


Figure 4.1: The Parallel DBF-SMM Architecture. Each AMU has a search window of w bytes that are one byte shifted with respect to each other.

assumption in Section 2.1. Then, the probability of a match in F_A and a no-match in F_F for a given AMU$_i$ is the same for all $i = 0 \dots s-1$ as in Eqn. (3.8).

The probability of F_A match and F_F no-match in exactly j AMUs is $\binom{s}{j} P_{A\overline{F}}^{j} \cdot (1 - P_{A\overline{F}})^{s-j}$ which is binomially distributed. In this case, EMU verification is performed serially for j AMU matches consuming a total time $j * t_{EMU}$.

Consequently, the average response time of the parallel architecture $T_{DBF-SMM-p}$ is expressed as follows:

$$\begin{aligned} T_{DBF-SMM-p} &= t_{AMU} + t_{AMU} \cdot E \cdot \textstyle\sum_{j=1}^{s} \binom{s}{j} \cdot {P_{A\overline{F}}}^{j} \cdot (1 - P_{A\overline{F}})^{s-j} \cdot j \\ &= t_{AMU} \cdot [1 + E \cdot s \cdot P_{A\overline{F}}] \\ &= t_{AMU} \cdot [1 + E \cdot s \cdot (P_{\mathcal{S}} \cdot (1-\alpha) + (1 - P_{\mathcal{S}}) \cdot Pfp_A)] \end{aligned} \tag{4.1}$$

The respective average response times of parallel DBF-SMM under $P_{\mathcal{S}} \approx 0$ and $P_{\mathcal{S}} \approx 1$ are as follows by Equations (3.9) and (3.10):

$$T_{DBF-SMM-p}^{P_{\mathcal{S}} \approx 0} = t_{AMU} \cdot [1 + E \cdot s \cdot Pfp_A] \tag{4.2}$$

$$T_{DBF-SMM-p}^{P_{\mathcal{S}} \approx 1} = t_{AMU} \cdot [1 + E \cdot s \cdot (1-\alpha)] \tag{4.3}$$

Here we would like to note that the average response time for a single query is longer for the parallel implementation when compared to the single AMU implementation because of the possible sequential EMU verifications.

Both Pfp_F and Pfp_A are defined per filter per query. On the one hand, Pfp_As of all F_As have a cumulative effect on $T_{DBF-SMM-p}$ because of the sequential EMU verification when there are multiple AMU outcomes with F_A match and F_F no match. On the other hand, the number of queries executed on the input string is the same for the single and parallel AMU implementations. Hence, Pfp_F does not accumulate and its effect is the same as the single AMU implementation.

CHAPTER 5

DBF-SMM IN PRACTICE

We assume that $\mathcal{S}$ is the most recent and complete set of strings. Accordingly, any new positive strings are added to $\mathcal{S}$ and stored in F_A.

The response time of DBF-SMM is at its minimum and it is not affected by the $P_{\mathcal{S}}$ when $\alpha = 1$ as discussed in Section 3.4. Hence, it is desired to have α as close as possible to 1.

To this end, we suggest a two-component approach to construct $\mathcal{F}$. Let $\mathcal{F}_{frequent} \subset \mathcal{S}$ be *a* set of strings that are known to appear frequently over long intervals of time. Let $\mathcal{F}_{dynamic} \subset \mathcal{S}$ be an additional *disjoint* set of strings whose content dynamically changes according to the recent match results. Then, $\mathcal{F} = \mathcal{F}_{frequent} \cup \mathcal{F}_{dynamic}$.

Accordingly, we implement F_F with two vectors; $v_{F\_frequent}$ and $v_{F\_dynamic}$ with $m_{F\_frequent}$ and $m_{F\_dynamic}$ bits respectively. Initially, $\mathcal{F}_{frequent}$ is stored in $v_{F\_frequent}$ and $v_{F\_dynamic}$ is reset. $v_{F\_frequent}$ stays constant during the operation and $\mathcal{F}_{dynamic}$ is continuously updated with selected strings $S_j \in \mathcal{S}$ where $S_j \notin \mathcal{F}_{frequent}$.

The implementation of the DBF-SMM in Fig. 3.1 is updated as shown on Fig. 5.1. The hash functions map the incoming string to v_A, $v_{F\_frequent}$ and $v_{F\_dynamic}$. F_F gives a match result if there is a match result from $v_{F\_frequent}$ or $v_{F\_dynamic}$. Hence the *maximum* value for Pfp_F is $Pfp_{F\_frequent} + Pfp_{F\_dynamic}$ where $Pfp_{F\_frequent}$ and $Pfp_{F\_dynamic}$ are computed according to Eqn. (2.2). The values for $|\mathcal{F}_{frequent}|$, $|\mathcal{F}_{dynamic}|$, $m_{F\_frequent}$ and $m_{F\_dynamic}$ should be selected such that $Pfp_{F\_frequent} + Pfp_{F\_dynamic} \leq Pfp_{F\_max}$ where $Pfp_{F\_max} = 2 \cdot 10^{-14}$ as defined in Section 3.5.

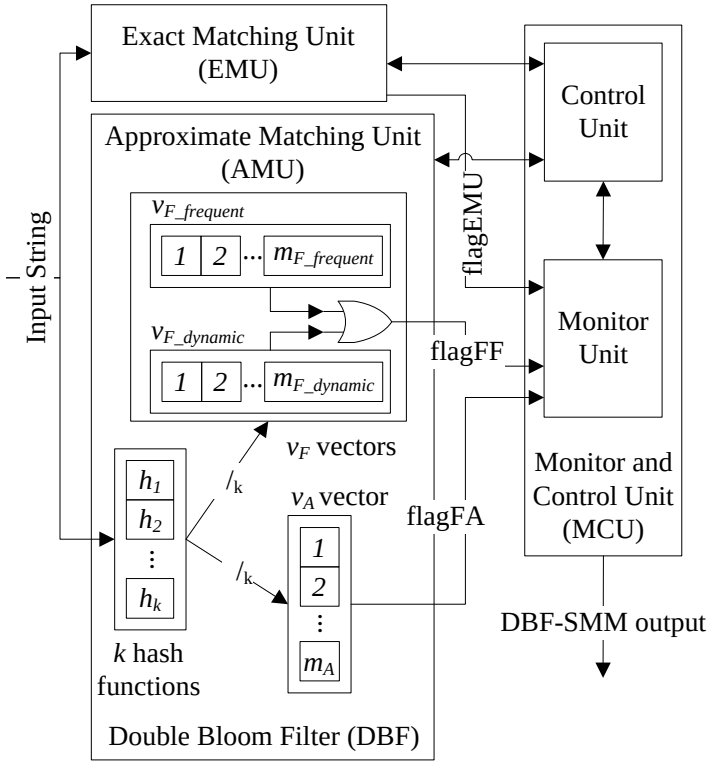


Figure 5.1: DBF-SMM with Dynamic Updates.

Here, we would like to note that the selection of $|\mathcal{F}_{frequent}|$ and $|\mathcal{F}_{dynamic}|$ depends on the hardware resources and the selection of their optimal values is not in the scope of this thesis. We suggest $|\mathcal{F}_{frequent}| = |\mathcal{F}_{dynamic}|$ which yields $m_{F\_frequent} = m_{F\_dynamic}$. Then, $Pfp_{F\_frequent} = Pfp_{F\_dynamic} = Pfp_{F\_max}/2 = 10^{-14}$.

We suggest that $v_{F\_dynamic}$ stores the most recent true positive strings that do not exist in the pre-determined $\mathcal{F}_{frequent}$ similar to a cache. This approach is particularly appropriate for the intrusion detection systems where the SMM checks the incoming packets against $\mathcal{S}$ that contains the attack signatures. Temporal locality of attacks is observed in distributed denial of service attacks and brute force attacks. Previous works such as [30] notes that SNORT fails under such *burst attacks*, i.e. it drops the 80% of the incoming packets against just 14 types of attack rules at 2Mbps rate. We note that, this is one way of determining the $v_{F\_dynamic}$ content and it is possible to define other management policies for $v_{F\_dynamic}$.

To this end, we call the matches that are positive $\mathcal{F}_{frequent}$ as dynamic matches.

We employ a Dynamic Match Counter (DMC) that continuously tracks the dynamic match rate (DMR). When DMR is below a given threshold, $v_{F\_dynamic}$ is cleared to prevent the accumulation of old matching strings which are not relevant recently.

Algorithm 1 describes the management of $v_{F\_dynamic}$ where the threshold value for DMR to clear $v_{F\_dynamic}$ is $1/c$. Note that a zero value for the DMC indicates that the DMR is below the $1/c$ match threshold per query.

Algorithm 1 $v_{F\_dynamic}$ management

1: Initially: $DMC = 0$, $v_{F\_dynamic}$ is cleared
2: For Each Queried String S_j:
3: **if** EMU reference returns true positive **then**
4: $DMC = DMC + c$, Store S_j in $v_{F\_dynamic}$
5: **end if**
6: **if** $v_{F\_dynamic}$ or $v_{F\_frequent}$ has a match **then**
7: $DMC = DMC + c$
8: **else**
9: $DMC = max(0, DMC - 1)$
10: **end if**
11: **if** $DMC = 0$ **then**
12: Clear $v_{F\_dynamic}$
13: **end if**

Fig.5.2 illustrates a simple example with $c = 5$. At time $t = 1$, the DMC is incremented because of a dynamic match (true positive match that is found in $v_{F\_dynamic}$ or $v_{F\_frequent}$ or verified by the EMU). When there are no dynamic matches in the consecutive queries DMC value is decremented for each query until a second dynamic match occurs at time $t = 5$. There are no dynamic matches after $t = 5$. DMC value decreases to zero at time $t = 11$ and the DMR value drops down to $1/c$ match per query clearing $v_{F\_dynamic}$.

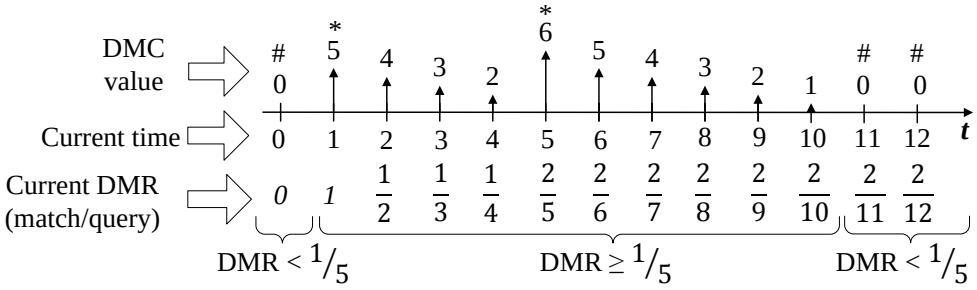


Figure 5.2: DMC is incremented by $c = 5$ because of a dynamic match denoted by * symbol. The DMC is decremented by 1 for queries that are not verified by the EMU to be a true positive match or unmatch cases for both $v_{F\_dynamic}$ and $v_{F\_frequent}$. When the DMC value drops down to the rate of $1/c$ match per query, the $v_{F\_dynamic}$ is cleared, which is depicted by #.

CHAPTER 6

DBF-SMM EVALUATION UNDER DIFFERENT DESIGN PARAMETERS

We evaluate the response time of both single AMU and parallel AMU implementations of our proposed DBF-SMM under different cases which are defined by the values of the design parameters and hit probabilities α.

To this end, we first evaluate the values of the response time expressions (3.2), (3.11) and (4.1) for these cases. Next, we implement selected cases on FPGA to demonstrate the feasibility of DBF-SMM implementation and provide a realistic analysis of the hardware resource consumption.

6.1 Response Time Evaluation Under Different Design Parameters

Without any loss of generality, we select $m_F = m_A$ for simplicity and partition this m_F evenly such that $m_{F\_frequent} = m_{F\_dynamic} = \left\lceil \frac{m_F}{2} \right\rceil$. Accordingly, $|\mathcal{F}_{frequent}| = |\mathcal{F}_{dynamic}|$. In Tables 6.1 and 6.2 m_{total} denotes the total memory requirement and $|\mathcal{F}| = |\mathcal{F}_{frequent}| + |\mathcal{F}_{dynamic}|$.

We enumerate a list of cases in Table 6.1 that represent a combination of the design parameters together with $|\mathcal{S}|$ and the EMU slow-down factor E as inputs.

The SMMs are frequently used in intrusion detection systems, hence, we chose a popular string matching based intrusion detection tool SNORT [5] as our reference. We analyzed the rule set of SNORT v2.9 and the length distribution of the signature strings is depicted in Fig. 6.1. There are 4095 content based rules represented as

strings and their lengths range from 1 byte to 970 bytes. The average string length is approximately 30 bytes and the volume of the overall database is 120k bytes.We select $|\mathcal{S}| = 5500$ and $|\mathcal{S}| = 7000$ assuming the expansion of the rule set of SNORT v2.9. [29] reports an EMU frequency of 59.24 MHz for typical software-based SNORT applications on a commodity processor and a $t_{AMU} = 3.225$ ns with 310MHz clock frequency yielding $E = 5$. Case 1 in Table 6.1 represents a baseline with $|\mathcal{S}| = 5500$ and $E = 5$.

The maximum false positive probabilities are defined as $Pfp_{A\_max} = 0.01/E$ and $Pfp_{F\_frequent} = Pfp_{F\_dynamic} = Pfp_{F\_max}/2 = 10^{-14}$. Accordingly, we select the number of hash functions $k = \lceil k_{opt} \rceil$ and determine $m_A = \lceil k \cdot |\mathcal{S}| \,/\, \ln 2 \rceil$ according to Equation (2.3). For the computed k and given $Pfp_{F\_frequent} = Pfp_{F\_dynamic}$, we compute the ratio of the number of strings to the bit vector size for both $F\_frequent = F\_dynamic$ according to Eqn.(2.5).

Next, we compute the cumulative false positive probability $Pfp_F = Pfp_{F\_frequent} + Pfp_{F\_dynamic}$ with the selected parameters to ensure $Pfp_F < Pfp_{F\_max}$ is satisfied. Similarly, we compute Pfp_A that is achieved with the selected parameters to show that $Pfp_A < Pfp_{A\_max}$.

Lastly we compute the response times under $P_{\mathcal{S}} \approx 1$ with $\alpha = 1$, $\alpha = 0.8$ and $\alpha = 0.4$ as in Eqn. (3.15) and compare it to an SBF-SMM with the same t_{AMU}, E and false positive probability. We provide the required amount of memory for the vector in SBF-SMM for completeness as $m_{SBF-SMM}$. Note that $T^{P_{\mathcal{S}} \approx 1}_{DBF-SMM} = T^{P_{\mathcal{S}} \approx 1}_{SBF-SMM}$ when $\alpha = 0$ as stated in Section 3.4.

Note that the response times are expressed in terms of t_{AMU} in Tables 6.1 and 6.2.

Next, we compute the design parameter and performance bound values for the parallel implementation of the base Case 1 with $s = 2$, $s = 4$ and $s = 8$ as shown in Table 6.2.

Similar to the single AMU implementation we define Pfp_{A_max-p} for Pfp_A to limit the penalty of false positives when $P_{\mathcal{S}} \approx 0$. To this end, $Pfp_{A\_max-p} = 0.01/s \cdot E$ by Equation (4.2).

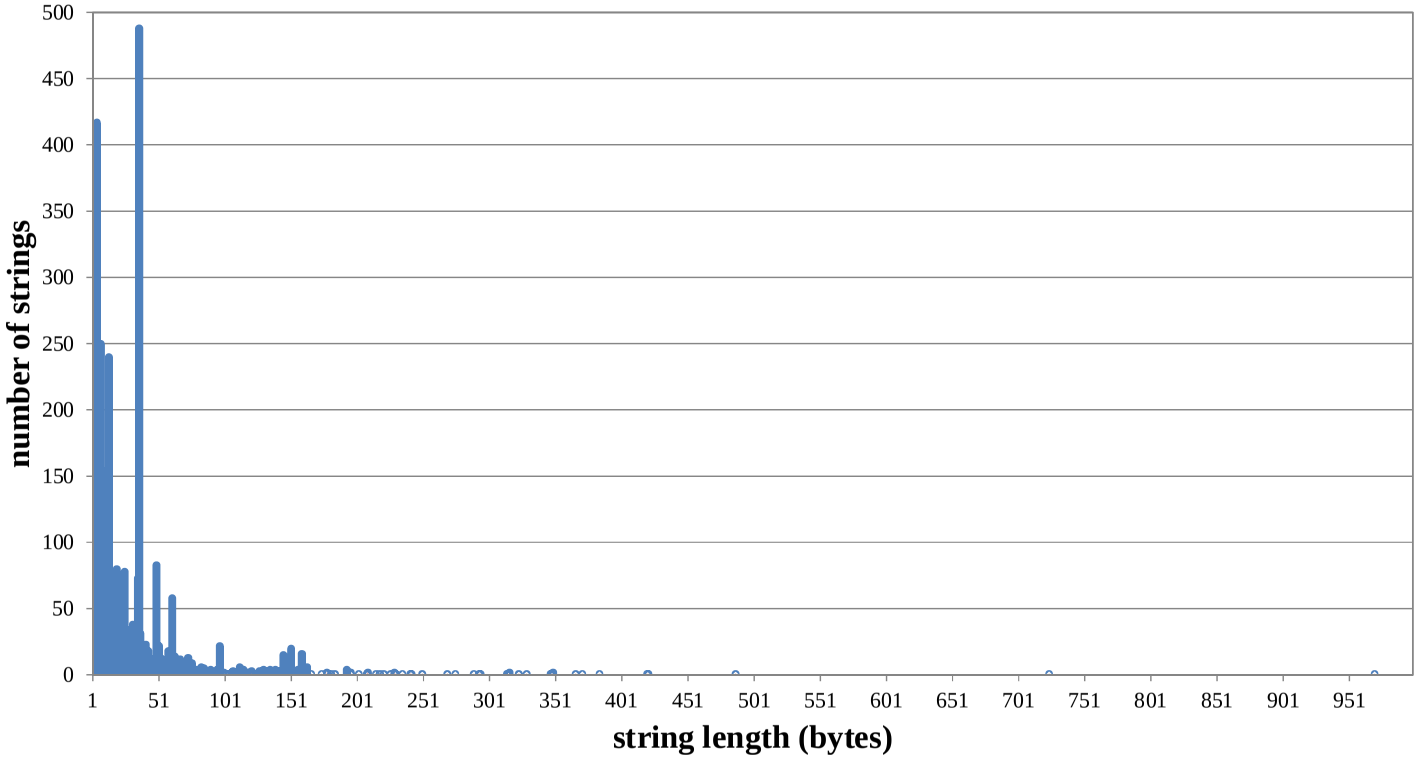


Figure 6.1: The length distribution of the signature strings in SNORT v2.9 database.

Table 6.1: Evaluation cases for the single AMU DBF-SMM.

	Case 1	Case 2	Case 3	Case 4
E	5	5	10	10
$\|\mathcal{S}\|$	5500	7000	5500	7000
$\|\mathcal{F}\|$	224	284	324	410
m_{total} (kbits)	144	182	160	202
k	9	9	10	10
Pfp_{A_max}	$2 \cdot 10^{-3}$	$2 \cdot 10^{-3}$	10^{-3}	10^{-3}
Pfp_A	$1.90 \cdot 10^{-3}$	$1.90 \cdot 10^{-3}$	$0.92 \cdot 10^{-3}$	$0.98 \cdot 10^{-3}$
Pfp_F	$1.87 \cdot 10^{-14}$	$1.92 \cdot 10^{-14}$	$1.94 \cdot 10^{-14}$	$1.99 \cdot 10^{-14}$
$T^{P_S \approx 1}_{DBF-SMM}, \alpha = 1$	1	1	1	1
$T^{P_S \approx 1}_{DBF-SMM}, \alpha = 0.8$	2	2	3	3
$T^{P_S \approx 1}_{DBF-SMM}, \alpha = 0.4$	4	4	7	7
$T^{P_S \approx 1}_{SBF-SMM}$	6	6	11	11
$m_{SBF-SMM}$ (kbits)	72	91	80	101

We derived the response time expressions for SBF-SMM, DBF-SMM and the parallel implementation of DBF-SMM in the previous sections. The corresponding scan rates as defined in Section 2.1 for these SMMs are;

$$\begin{aligned} R_{SBF-SMM} &= 8/T_{SBF-SMM} \text{ bps}, \\ R_{DBF-SMM} &= 8/T_{DBF-SMM} \text{ bps}, \\ R_{DBF-SMM-p} &= 8 \cdot s/T_{DBF-SMM-p} \text{ bps}. \end{aligned} \tag{6.1}$$

Accordingly, we present the response times for the parallel DBF-SMM implementations normalized by s for fair comparison in Table 6.2.

Table 6.2 shows the trade-offs between the increased memory (m_{total}) and logic (k_{total}) resource consumption and decreased response times when DBF-SMM is implemented in parallel.

Table 6.2: Evaluation cases for the parallel AMU DBF-SMM.

	s=2	s=4	s=8
E	5	5	5
$\|\mathcal{S}\|$	5500	5500	5500
$\|\mathcal{F}\|$	324	438	564
m_{total} (kbits)	320	704	1536
k per AMU	10	11	12
k_{total}	20	44	96
Pfp_{A_max}	10^{-3}	$5 \cdot 10^{-4}$	$2.5 \cdot 10^{-4}$
Pfp_A	$0.92 \cdot 10^{-3}$	$4.59 \cdot 10^{-4}$	$2.28 \cdot 10^{-4}$
Pfp_F	$1.94 \cdot 10^{-14}$	$1.96 \cdot 10^{-14}$	$1.98 \cdot 10^{-14}$
$T^{P_S \approx 1}_{DBF-SMM-p}/s, \alpha = 1$	0.5	0.25	0.125
$T^{P_S \approx 1}_{DBF-SMM-p}/s, \alpha = 0.8$	1.5	1.25	1.125
$T^{P_S \approx 1}_{DBF-SMM-p}/s, \alpha = 0.4$	3.5	3.25	3.125
$T^{P_S \approx 1}_{SBF-SMM-p}/s$	5.5	5.25	5.125

CHAPTER 7

FPGA IMPLEMENTATION OF DBF-SMM

7.1 Introduction

Field programmable gate arrays (FPGAs) are semiconductor devices designed to be configured in the field after manufacturing. They mainly consist of configurable logic blocks (CLBs) that are connected via programmable interconnects. FPGAs (especially RAM-based types) can be reprogrammed over and over again to perform required changing functionality. Reprogrammability, lack of manufacturing cost and short time-to-market are three major advantages of FPGAs (except one-time programmable types) with respect to Application Specific Integrated Circuits (ASICs) which are manufactured to perform dedicated functions. Moreover, different than processors, FPGAs have intrinsic parallelism. Therefore, multiple processing tasks can be performed independently and concurrently by dedicated CLBs of an FPGA. Specific applications of FPGAs include ASIC prototyping, computer hardware emulation, digital signal processing, servers, routers, switches, network intrusion detection systems, etc.

In this study, after the design, analysis and evaluation of the DBF-SMM, an FPGA design platform is employed to see the feasibility of the proposed architecture, which include multiple hash functions running concurrently. For this purpose, single and parallel DBF-SMM architectures are implemented on a real FPGA device to achieve hardware resource consumptions and clock timings. However, the verification of the DBF-SMM functionality is performed in SystemC as given in Chapter 8. This is because FPGA design tools poorly support such a complex verification platform.

7.2 VHDL Description of DBF-SMM

To define the behavior of FPGA modules, FPGA vendors provide platforms that support hardware description languages (HDL) and/or schematic editors. Due to high complexity of AMU structure, we prefer to use VHDL (**V**ery High Speed Integrated Circuit **HDL**). However, a schematic tool that illustrates the generated logic are utilized to check the definition and the synthesis of the VHDL descriptions as seen in Fig.7.1. Although SystemC also has capability of hardware description level similar to VHDL, we do not prefer it due to poor support of FPGA vendors.

The fast components of DBF-SMM are the AMU and MCU while the EMU is assumed to be a slow and possibly software component. To this end, we implement the AMU and MCU shown in Figure 5.1 on Xilinx XC7VX550T-3FFG1158 FPGA [6] by using ISE 14.5 [7] software, which is illustrated in Fig.7.1 where a part of the synthesized AMU block is being depicted. The selected FPGA device consists of 86600 logic slices and 2360 Block RAMs each of which has a size of 18kbits and supports 28 Gbps transceivers.

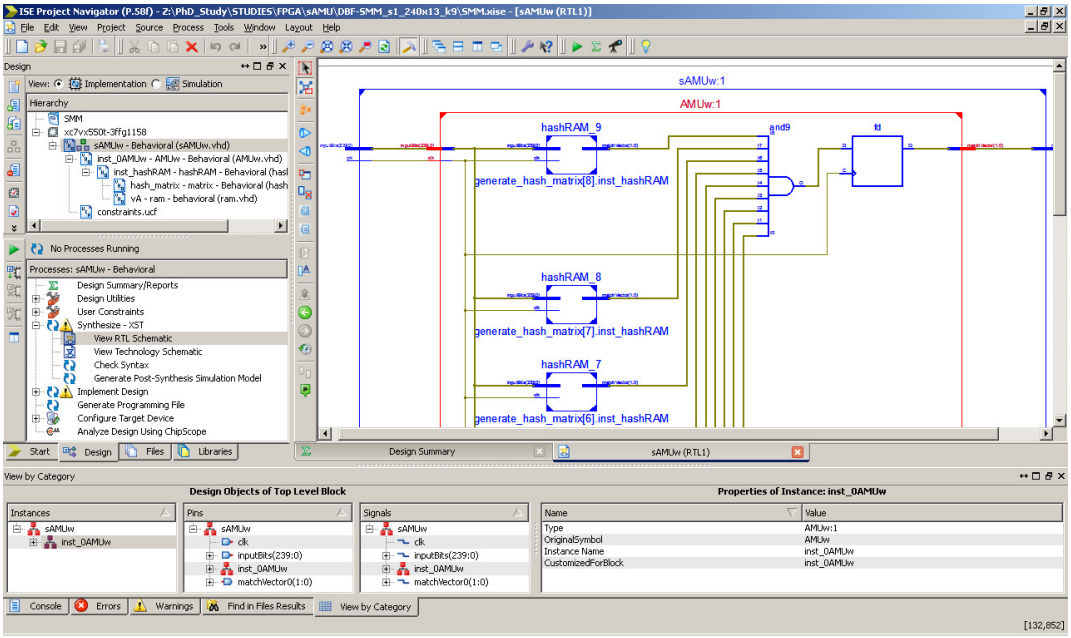


Figure 7.1: Synthesized AMU circuit of DBF-SMM Architecture.

The string length that is defined in Section 3.6 is considered as $w = 240$ bits. Each

AMU query is completed in a single clock cycle, hence, t_{AMU} = Clock Period.

One port of each Block RAM (BRAM) is assigned to only one hash function [16] as indicated in Fig.7.2 and the other port is reserved in order to externally update the RAM content on the fly.

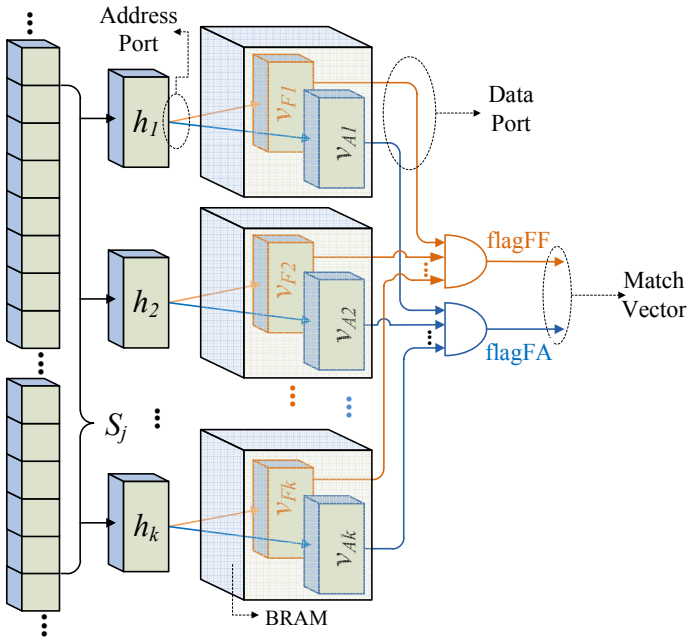


Figure 7.2: Implementation of AMU circuit on an FPGA.

7.3 FPGA Implementation Results

In Table 7.1 we summarized the resource consumptions and depict the achieved clock periods for the base Case 1 given in Table 6.1 and all of the three parallel implementations given in Table 6.2.

We normalize the reported EMU frequency of 59.24 MHz for typical software-based SNORT applications on a commodity processor as in [29] to the current day by assuming processor frequencies almost doubled from 1.5GHz in 2007 to 3GHz in 2014. Hence we assume the EMU frequency of 100MHz. Accordingly, the resulting scan rate estimates under $P_S \approx 1$ and different values of α are also presented in Table 7.1.

Each hash function maps 8k locations in the associated memory; therefore, for each

Table 7.1: FPGA implementation results for DBF-SMM.

	s=1	s=2	s=4	s=8
m_{total} **(kbits)**	144	320	704	1536
k_{total}	9	20	44	96
Block RAM	9	20	44	96
Slices	1072	1480	3031	5836
Clock Period (ns)	2.98	3.31	3.60	6.98
Scan Rate (Gbps), $\alpha = 0$	0.47	0.59	0.68	0.74
Scan Rate (Gbps), $\alpha = 0.4$	0.62	0.84	1.03	1.16
Scan Rate (Gbps), $\alpha = 0.8$	0.89	1.46	2.14	2.79
Scan Rate (Gbps), $\alpha = 1.0$	1.15	2.29	4.59	9.17

Block RAM, 8kbit is consumed for v_A for both SMMs; however, extra 8kbits are occupied by the v_F vector in DBF-SMM. Hence, the total number of hash functions is equal to the total number of employed Block RAMs.

An increase in the clock period is expected as the design grows due to increase in the routing (or interconnect) delays between the logic blocks. The logic consumption for different s values increases with a factor that is less than s. The matrix that defines a given hash function as described in Section 3.6 is constructed randomly. If there are overlaps in these elements of matrices for different hash functions, the FPGA optimization tool may only generate one shared instance which conserves certain amount of logic source. Furthermore, the FPGA optimization tool may generate better implementations as the circuit grows with increasing s.

As a conclusion, the implementation results show that it is possible to implement DBF-SMM with 8 parallel AMUs on XC7VX550T-3FFG1158 device with a 6.7% logic and 4.1% memory resource utilization achieving 6.98ns clock period. Therefore, the proposed DBF-SMM can be realized in hardware in order to match strings at 9.1 Gbps average scan rate by storing 5500 strings where each of which is 30-byte long. This scan rate can also be sustained if the strings stored in F_F is limited to $|F| < 564$.

CHAPTER 8

SYSTEMC IMPLEMENTATION OF DBF-SMM

8.1 Introduction

SystemC is a modeling and verification environment, which provides a wide range of abstraction levels from system-level to register-transfer level (RTL). The designer describes the system by using SystemC methodology, which is defined by a C++ class library entirely built on top of the standard object-oriented C++ language with an event-driven simulation kernel. Following the modeling process, the model can be verified in order to see whether it works as expected or not with the opportunity of utilizing capabilities of C++ language. SystemC was defined by the Open SystemC Initiative (OSCI), and has been approved by the Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA) as IEEE 1666-2011 [35]. In December 2011 Accellera and Open SystemC Initiative (OSCI) approved their merger, adopting the name Accellera Systems Initiative, which is an independent, not-for-profit organization dedicated to system-level design, modeling, and verification standards [1].

In this thesis, a model of the DBF-SMM is implemented by employing the classical hardware modeling capability of SystemC language (RTL). The functionality of the model is verified thoroughly by employing input strings having different characteristics. The main motivation for modeling the DBF-SMM in SystemC is that the verification process can be carried out easily and quickly.

SystemC 2.3.0 release is used in all of the SystemC implementation parts, which is employed with the Microsoft Visual Studio 2010 Ultimate (Version 10.0.30319.1) environment for the implementation and verification stages of the model.

8.2 SystemC Model of DBF-SMM

Basically, SystemC simulator has two main phases: (1) *elaboration* and (2) *execution*, which run respectively. During the elaboration phase, needed processes due to be prepared for the execution phase are performed such as the initialization of the data structures, the establishment of the connectivities between modules, etc. SystemC simulation kernel controls the execution phase, which runs the processes in such a way that they seem running concurrently.

The SystemC model of the DBF-SMM is indicated in Fig. 8.1. There are three main parts; (1) *AMU*, (2) *Stimulus* and (3) *Monitor*.

AMU consists of k hash functions $\mathcal{H} = \{h_1, h_2, \cdots, h_k\}$, where each of which addresses a Dual Port RAM. Data ports of RAMs are logically ANDed to indicate that queried input string $S_j \in \mathcal{S}$ or not. During the elaboration phase, string database $\mathcal{S}$ is *stored* in the Bloom Filter F_A by calculating k hash values for each element of $S_j \in \mathcal{S}$ and setting the location of $h_i(S_j)$ in the associated Dual Port RAM RAM_i. Besides, each input string $S_j \in P$ is *queried* by AMU in the execution phase. For this purpose, k hash functions $h_i(S_j)$ are calculated and each of them addresses the associated RAM_i locations, i.e. $RAM_i(h_i(S_j))$. Note that each RAM cell has 2 bits; one is dedicated for the match flag of F_A and the other is for F_F. Each RAM output is logically ANDed to achieved value of the match vector having also 2 bits.

In order to keep the design as simple as possible and decease the simulation time, MCU is implemented inside Stimulus. By this way, we keep the complexity inside and eliminate the time spend due to high number of transmission channel events. Due to increase the simulation time, the exact matching unit is also not implemented. Instead, the input string traffic P is controlled by Stimulus in such way that each input string S_j queried by the SMM is know that whether it is a member of the string database $\mathcal{S}$ or not, which is indicated by true positive variable tp. For this purpose, Stimulus selects a string S_j from a set of innocent input strings $S_i \notin \mathcal{S}$ or from the string database $\mathcal{S}$, depending on the aimed input string characteristics. Stimulus unit controls the execution of the DBF-SMM by generating write enable (we) and erase enable (ee) control signals according to the match vector.

Throughout the execution of the model, the monitor unit prints the most significant variables on screen as a console output and writes more detailed information into log files to be inspected soon.

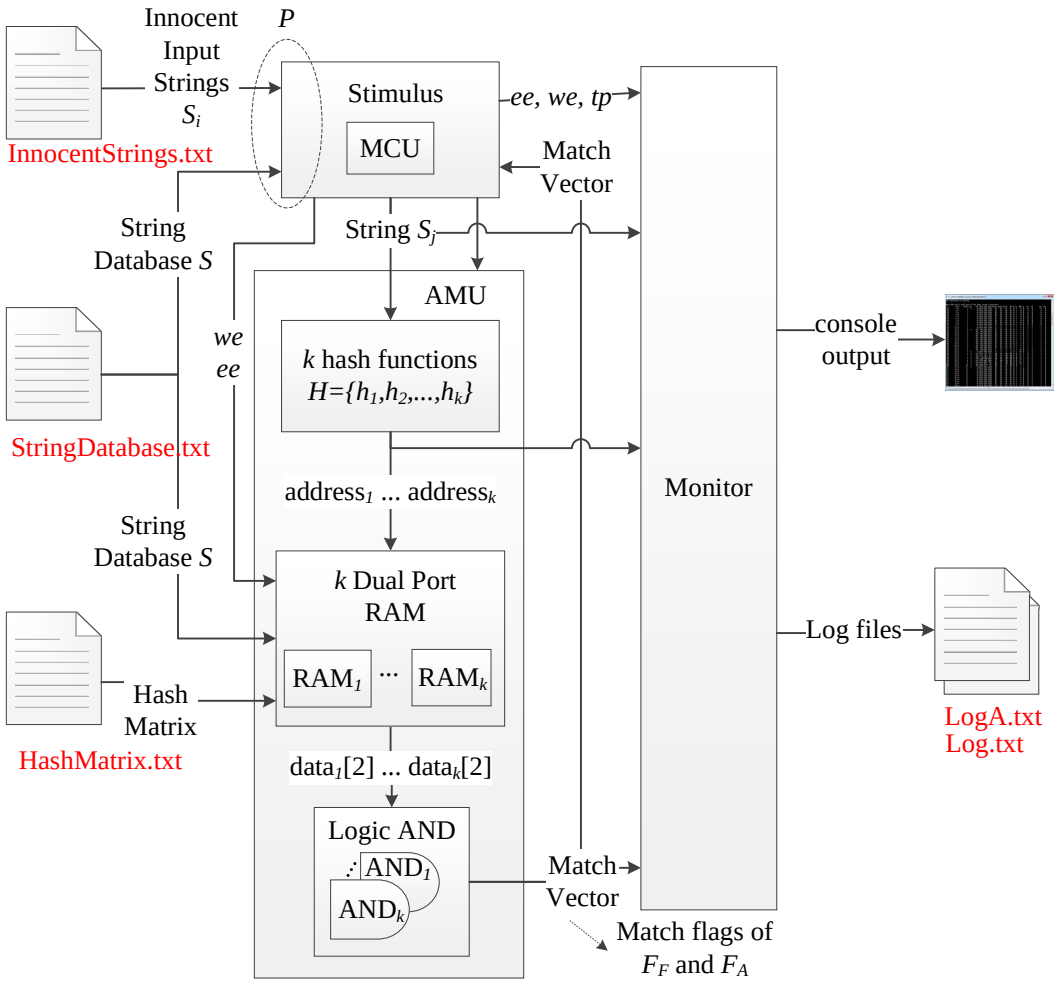
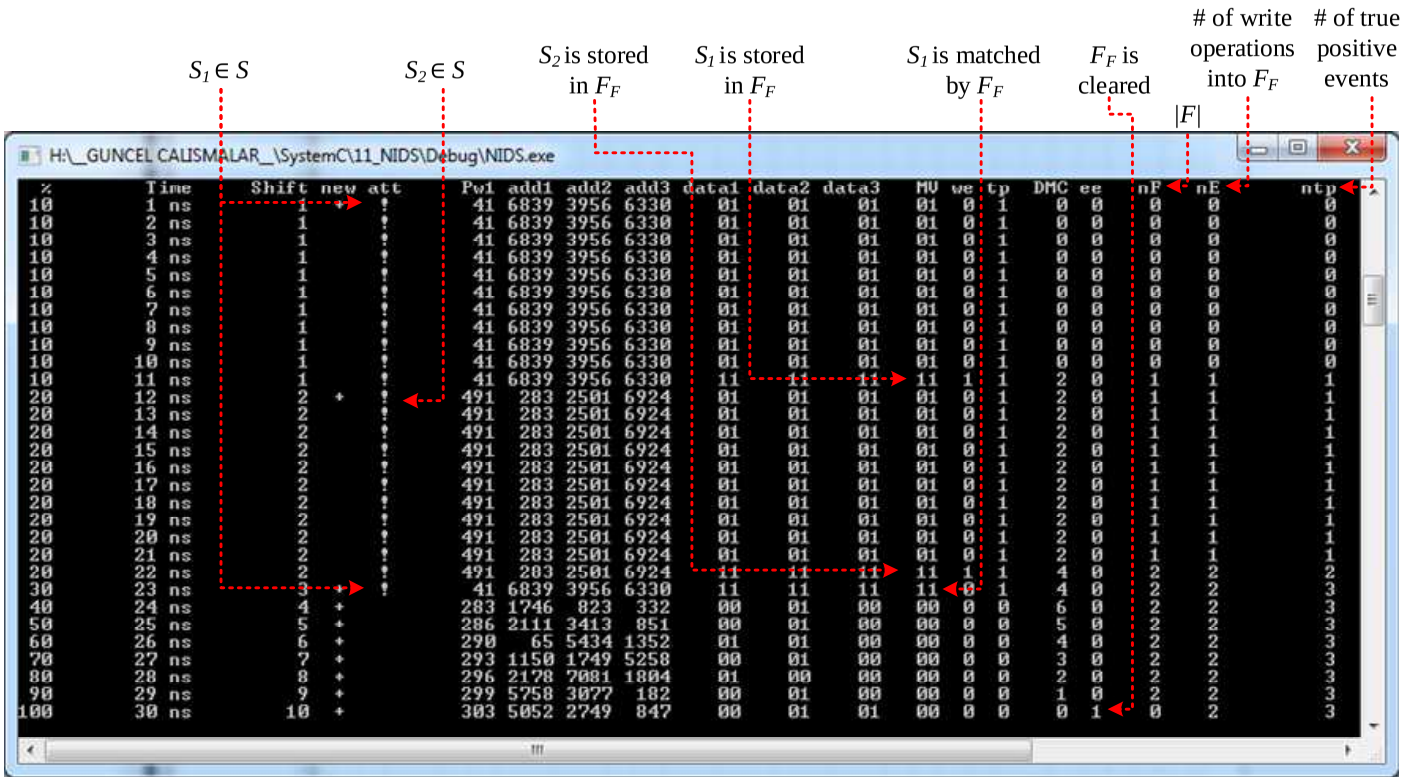


Figure 8.1: The SystemC Model of DBF-SMM.

Fig.8.2 exemplifies the execution of the model with DMC constant $c = 2$. $S_1 \in \mathcal{S}$ is matched by F_A at time 1 ns. Following the verification process, which lasts $E = 10$ cycles, S_1 is stored in F_F. The S_1 is matched by both F_F and F_A at time 23 ns and the verification is eliminated. True positive events are counted by the variable ntp and for each of these events DMC value is incremented by $c = 2$ as illustrated at time 11 ns, 22 ns and 24 ns. Differently, it is decremented by one for each false positive events. F_F is cleared by Stimulus at time 30 ns because DMC is zero.



%	Time	Shift	new	att	Pw	add1	add2	add3	data1	data2	data3	MV	we	tp	DMC	ee	nP	nE	ntp
10	1 ns	1	+	!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	2 ns	1		!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	3 ns	1		!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	4 ns	1		!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	5 ns	1		!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	6 ns	1		!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	7 ns	1		!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	8 ns	1		!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	9 ns	1		!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	10 ns	1		!	41	6839	3956	6330	01	01	01	01	0	1	0	0	0	0	0
10	11 ns	1		!	41	6839	3956	6330	11	11	11	11	1	1	2	0	1	1	1
20	12 ns	2	+	!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	13 ns	2		!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	14 ns	2		!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	15 ns	2		!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	16 ns	2		!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	17 ns	2		!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	18 ns	2		!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	19 ns	2		!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	20 ns	2		!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	21 ns	2		!	491	283	2501	6924	01	01	01	01	0	1	2	0	1	1	1
20	22 ns	2		!	491	283	2501	6924	11	11	11	11	1	1	4	0	2	2	2
30	23 ns	3	+	!	41	6839	3956	6330	11	11	11	11	0	1	4	0	2	2	3
40	24 ns	4	+		283	1746	823	332	00	01	00	00	0	0	6	0	2	2	3
50	25 ns	5	+		286	2111	3413	851	00	01	00	00	0	0	5	0	2	2	3
60	26 ns	6	+		290	65	5434	1352	01	01	00	00	0	0	4	0	2	2	3
70	27 ns	7	+		293	1150	1749	5258	00	01	00	00	0	0	3	0	2	2	3
80	28 ns	8	+		296	2178	7081	1804	01	00	00	00	0	0	2	0	2	2	3
90	29 ns	9	+		299	5758	3077	182	00	01	00	00	0	0	1	0	2	2	3
100	30 ns	10	+		303	5052	2749	847	00	01	01	00	0	0	0	1	0	2	3

Figure 8.2: An example for the execution of the SystemC model.

8.3 Verification of the DBF-SMM Model

To verify the model, we first test the false positive probability of the filter F_A. For this purpose, a string set $\mathcal{S}$ consisting of 5500 strings where each of which are $w = 30$ bytes long is randomly generated. $k = 10$ hash functions are used and each of them maps 8192 locations in RAM. The theoretical value of the false positive rate calculated according to Eqn. 2.5 is $Pfp_A = 7.82 \cdot 10^{-4}$. The simulation is run 10 times, where each run includes 10^6 AMU queries. The average false positive rate of the simulation is $7.96 \cdot 10^{-4}$, where it is 1.79% worse than the theoretical value. The slight variation between these values may be because of the $rand()$ function given in C++ library.

Next, a DBF-SMM whose design parameters are based on the case-3 given in Table 6.1 is implemented in SystemC environment. For the input string traffic, each combination of $P_{\mathcal{S}} = \{0, 0.4, 0.8, 1\}$ and $\alpha = \{0.4, 0.8, 1\}$ are utilized. Therefore, 10 different kinds of traffic are generated (Note that $\alpha > 0$ values are meaningless for $P_{\mathcal{S}} = 0$). Each traffic type is simulated with 10 times, where each simulation includes 10^6 clock cycles. The simulation results are summarized in Fig.8.3, and they are consistent with the theoretical values previously given in Table 6.1 and Eqn.3.11. Therefore, the SystemC model verifies the functionality of the proposed DBF-SMM.

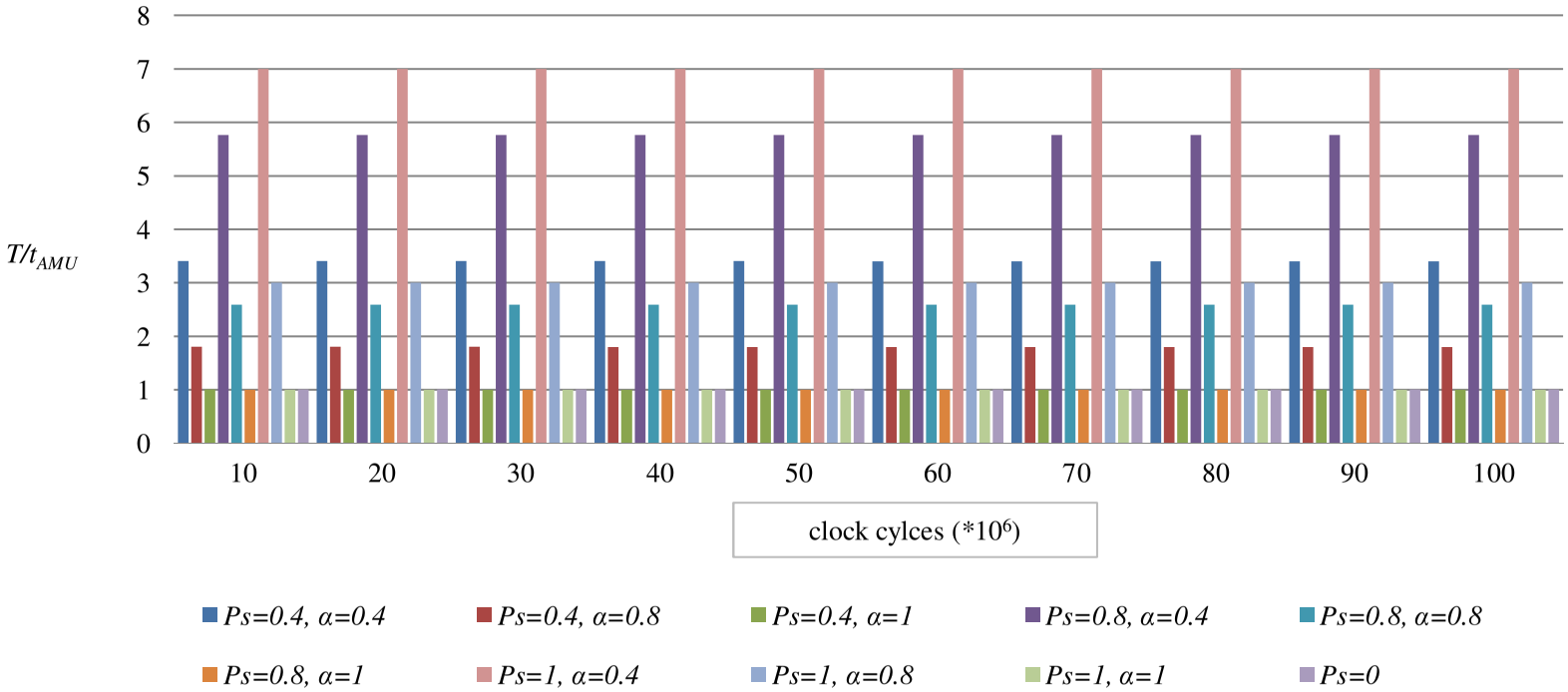


Figure 8.3: Simulation results of the SystemC model evaluated with 10 different traffics.

CHAPTER 9

VARIABLE SIZE STRING MATCHING WITH AUTOMATA

The design of DBF-SMM presented, analyzed and evaluated above assumes that all strings in the database $\mathcal{S}$ are fixed-length of w bytes. In this chapter, we present the extension of this machine to store also different size strings in order to provide arbitrarily-length string matching process.

Representing arbitrarily long strings with BFs may be performed in two ways:

- A classic BF [10] stores only a specific length of strings. Hence, in order to support arbitrarily-long strings, each length of strings can be stored into a specific BF [15, 16]. Therefore, a string set consisting of n number of different lengths requires a set of n number of Bloom filters, each of which is dedicated to a specific length of strings. In this concept, there are two remarkable problems. On one hand, the final machine may not be implementable for string sets consisting of large number of long strings due to hardware resource limitations and accumulation of logic delays. For instance, 171 number of BFs is needed whose lengths range from a single byte to 970 bytes to store the string set of SNORT v2.9. On the other hand, the number of BFs and their window sizes are also strongly dependent on the string set. Hence, frequently changing sets are not proper to be stored, because these types of sets necessitate frequent re-construction of the hardware logic, which is a very demanding and long process.

- Divide-and-conquer approach can be applied in order to handle different length strings. To do this, short strings, whose lengths are less than or equal to w bytes,

can be stored into a set of w BFs, each of which is dedicated to a specific length ranging from 1 to w bytes. Besides, long strings, whose lengths are longer than w bytes, can be divided into substrings of w-bytes in length and each of these w-byte substrings can be stored into a specific BF, namely w-byte BF. The residual substrings can also be stored into appropriate BFs according to their lengths. In this approach, the first, the middle and the residual (last) substrings are called *prefix*-, *suffix*-, and *postfix*-substrings respectively. For instance, with $w = 3$, the string *metueee* is divided into a prefix, suffix and postfix substrings of *met*, *uee*, and *e*, respectively. The *met* and *uee* are stored into a 3-byte BF and the postfix-substring *e* is stored into a 1-byte BF. Therefore, to represent a set of string set, each substring must be stored. Next, at the query phase, the correct sequence of the substrings must also be tracked correctly, which can be performed by an automaton [16, 8]. Instead of feeding the strings into SMM byte-by-bye, dividing the strings into w substrings enables us to achieve w times speed-up [16]. However, each character in the input string should have the chance to be considered as the first character. This issue is called byte alignment problem and it can be eliminated with employing w SMM, each of which has one byte shifted window (The details are given in Section 10.1.2) [16]. In this study, we map any string set into well-known Aho-Corasick based DFA [8], which can match multiple strings at a time with deterministic rate of state transition per time. After that, the generated state transition conditions are stored into appropriate look-up tables and BF-based structures in order to achieve high efficiency in terms of memory usage. In addition, the false positive disadvantages of the BFs are also handled with the previously applied concept of double Bloom Filters.

Firstly, Section 9.1 defines w-byte Deterministic Finite Automaton and gives some of its hardware implementations. And then, the usage of Bloom Filters (BF-wDFA) in order to approximately and compactly implement DFAs are introduced. After that, a well-known string matching automaton, Aho-Corasick machine, is described with an example and some of its properties are given in Section 9.2. Following that, we map the well-known SNORT string set into Aho-Corasick based wDFA (AC-wDFA) in Section 9.3. The outcomes of this study are employed to support the properties given

in 9.2 and calculate the memory efficiencies of the proposed SMM architectures. Next, BF-wDFA structure is modified to realize AC-wDFA and then its improved version which implements multiple BFs are given in Section 10.1. The proposed machines are evaluated and their performance are summarized in Section 10.2. Lastly, the related works in the literature are mentioned at the end of the chapter.

9.1 Bloom Filter based w-byte Deterministic Finite Automaton

The main aim of the proposed BF-based Deterministic Finite Automaton is efficiently storing the large transition tables into the expensive but high-speed hardware resources. Bloom filters need to calculate k hash functions, therefore achieving short query times are not possible on general software-based implementations. On the other hand, implementing them on a specialized hardware, like FPGAs or ASICs, enables us to gain data-level parallelism.

This part starts by defining w-byte Deterministic Finite Automaton and its use in variable-size string matching and follows with their typical hardware implementations. Then, we introduce the usage of Bloom filters in order to approximately and compactly implement these automatons.

9.1.1 w-byte Deterministic Finite Automaton (wDFA)

9.1.1.1 Formal Definition of wDFA

A w-byte Deterministic Finite Automaton (wDFA) is represented by a pentuple,

$(Q, \Sigma^w, \Delta, q_0, G)$, where:

- $Q = \{q_0, q_1, q_2 \cdots q_i \cdots\}$ is a finite, non-empty ordered set of states. The ordering is inferred from the naming of the states and the operation of incrementing defined as $q_i + 1$ returns the next state in the order q_{i+1}.
- Σ^w is a finite, non-empty set of w-byte symbols representing automaton alphabet. Note that Σ^w is w power of an alphabet Σ that is a finite, non-empty set of

single byte symbols. In addition, ${P^w}_i$ depicts the current input symbol that is composed of concatenation of w bytes labeled as $c_1 \cdots c_w$, and ${P^w}_i \in \Sigma^w$.

- $\delta = \{\delta_{c,i}\}$ is a finite set of state transition conditions where $\delta_{c,i} = \{(q_c, {P^w}_i) : q_c \in Q$, ${P^w}_i \in \Sigma^w$ and $(q_c, {P^w}_i) \in Q\text{x}\Sigma^w\}$

- $\Delta = \{\Delta_{c,i,n}\}$ is a finite set of state transition rules where $\Delta = \{\delta \rightarrow Q$: $\Delta_{c,i,n} = (q_c, {P^w}_i, q_n)$, $q_n \in Q\}$.

- q_0 is the start (or initial) state.

- G is a set of accepting (or final) states, where $G \in Q$.

Most of the traditional DFAs are defined with w = 1, it means that they consume only one byte at a time. In other words, the input stream is processed using a window of eight bits and advancing this window one byte on the input during one operating cycle. On the other hand, a *multi-byte* DFA can be realized with $w > 1$ in order to speedup the automaton by consuming multiple bytes (please refer to Section 9.2.1 for more details).

The size of the symbol alphabet is $|\Sigma^w| = 2^{8 \cdot w}$ and the number of possible state transitions is $|\Delta| = |Q| \cdot |\Sigma^w|$, where $|Q|$ and w are the numbers of states and 8-bit ASCII characters in the input symbol, respectively. Note that $|\Sigma^w|$ grows exponentially as a function of w.

A w-byte Non-deterministic Finite Automaton (wNFA) can be represented by a similar pentuple as above but replacing the set of transition rules. Hereby, $\Delta_{wNFA} = \{\delta \rightarrow Q_n : (q_c, {P^w}_i, Q_n)$, where $q_c \in Q$, ${P^w}_i = c_1 \cdots c_w$ is input symbol and $Q_n \in 2^Q\}$. 2^Q is the *power set* which is a set of all subsets of Q. Hence an NFA can be at a set of states Q_n at a time where DFA can be only at a single state. In other words, due to achieve next state from a given state on a given input symbol, DFA requires only a single state transition; however, NFA can have zero or more.

Throughout this thesis, NFA and DFA are referred as *finite automaton (FA)* or just *automaton* and finite sets of state transition rules are referred as *transitions* for simplicity reasons.

We next define the $depth(q)$ function as the minimum number of transitions from the initial state q_0 to some state $q \in Q$. Accordingly, the set of states Q can be classified into subsets according to their depth values. Let Q_i represents a set of states whose minimum number of state transitions from q_0 is i, i.e. $Q_i = \{q : q \in Q \text{ and } depth(q) = i\}$. Note that zero-depth state is q_0, i.e. $Q_0 = \{q_0\}$.

A set of state transition rules Δ can also be classified into four categories:

- *Consecutive Transitions,* $\Delta_C = \{(q_c, P^w{}_i, q_n)$, where $q_n = q_c + 1\}$
- *Unconsecutive Transitions,* $\Delta_U = \{(q_c, P^w{}_i, q_n)$, where $q_n \neq q_c + 1\}$
- *Accept Transitions,* $\Delta_A = \{(q_c, P^w{}_i, q_n)$, where $q_n \in G\}$
- *Depth-i Transitions,* $\Delta_i = \{(q_c, P^w{}_i, q_n)$, where $q_n \in Q_i\}$

In like manner, the associated transition conditions can be defined as $\delta_C, \delta_U, \delta_A, \delta_i$, respectively.

The *consecutive transitions* are the state transitions from state q_c to the consecutive next states $q_c + 1$. On the contrary, *unconsecutive transitions* are the transitions to unconsecutive next states. Transitions from any state to one of the G states are *accepting transitions*. Likewise, transitions to a next state of depth-i are defined as *depth-i transitions*.

Note that there are some important relations between these transitions as follows:

- $\Delta_C \cap \Delta_0 = \phi$. The next state index $n \geq 0$.
- $\Delta_C \cap \Delta_U = \phi$. A transition can not be consecutive and unconsecutive at the same time, i.e. Δ_C and Δ_U are disjoint sets.
- $\Delta_0 \subseteq \Delta_U$. As a consequence of the above, all of the Δ_0 transitions are also Δ_U transitions.
- $(\Delta_C \cap \Delta_A) \cup (\Delta_U \cap \Delta_A) = \Delta_A$ i.e. any Δ_A transition is either Δ_C or Δ_U transition.
- $\Delta = \Delta_C \cup \Delta_U$.

The structure of wDFA can be illustrated as in Fig.9.1. The machine always begins the execution on the start/initial state q_0. It reads (or consumes) input symbols $P^w{}_i$ one by one. For each input symbol, a state transition rule δ determines the next state q_n which is associated with the current input symbol $P^w{}_i$ and the current state q_c. A delay or memory element holds the current state q_c and after a while, it updates the output with a next state q_n. The output of the machine is defined as accepting state code with an indicator of accepting state flag. This flag is logic 1 if $q_c \in G$, otherwise it is logic 0. Note that the output function depends only the current state q_c; therefore the machine corresponds to the Moore model. However, the output function can also be defined as a function of q_c and $P^w{}_i$ that corresponds to the Mealy model. After updating the current state and the output, a new operating cycle begins with reading a new input symbol.

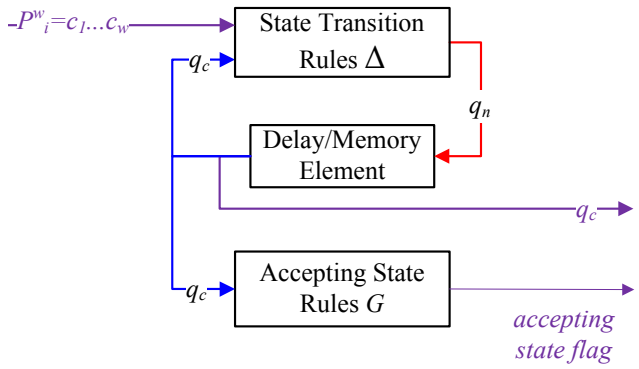


Figure 9.1: Structure of w-byte DFA (Moore Model).

wDFA is exemplified with a classic AC-based DFA ($w = 1$) representing a set of strings $\mathcal{S} = \{he, she, his, hers\}$. This machine is generated with AC algorithms given in [8], which are also summarized in 9.2. The state transition graph and state transition table are given in Fig.9.2 and Table 9.1 respectively.

It has 10 states $Q = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and 4 of them are accepting states $G = \{2, 5, 7, 9\}$. In Fig.9.2, solid vertices depicts to accepting states and each of them matches a non-empty subset of $\mathcal{S}$ respectively. Although the alphabet size of the DFA is $|\Sigma^w| = 2^8 = 256$, only 5 of them are employed by the string set. The number of possible state transitions is $|\Delta| = |Q| \cdot |\Sigma^w|$ = 2560, which are compactly illustrated as edges in Fig.9.2. The Δ_0 and Δ_1 transitions are emphasized as dashed and dotted lines

because they are responsible for large portion of the overall transitions ($|\Delta_0| = 2534$ and $|\Delta_1| = 15$). All of the transition types are given explicitly in Table 9.1. In Section 10.1, we map these transitions into expensive hardware blocks in a proper manner in order to achieve high memory efficiency.

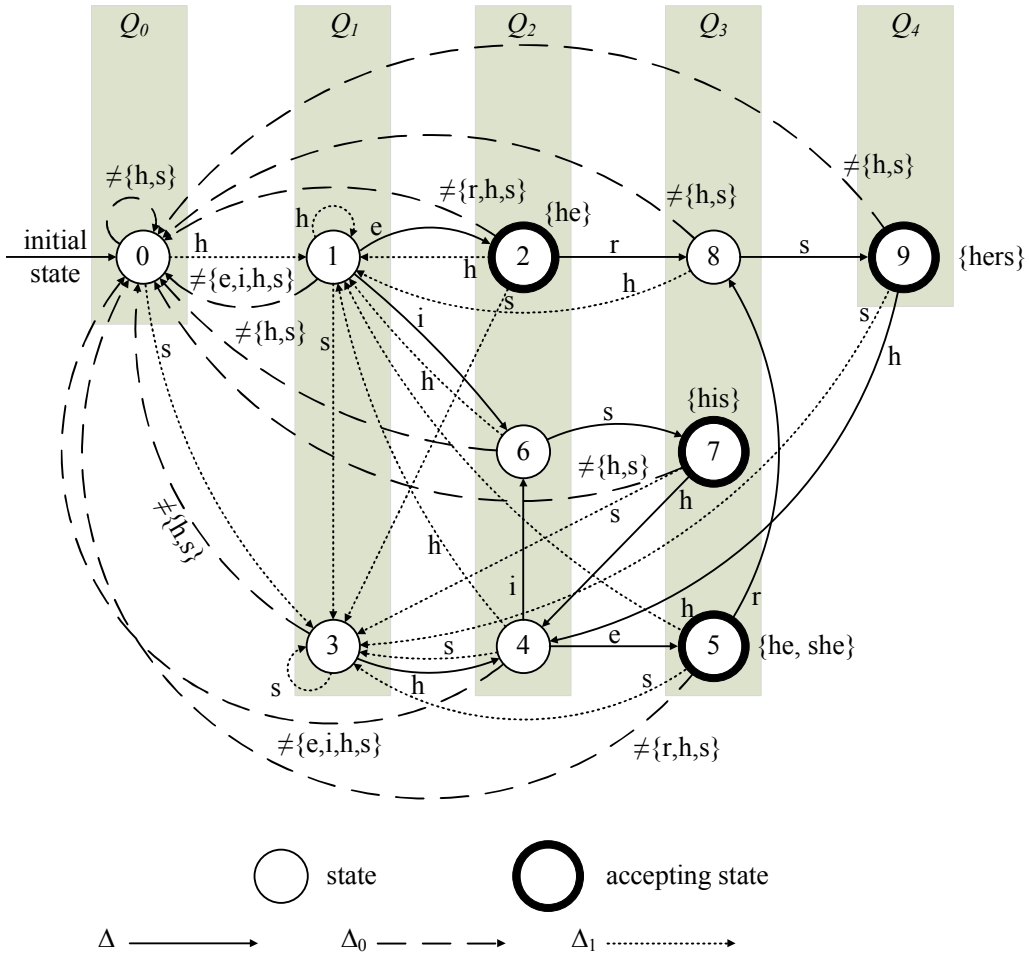


Figure 9.2: State transition graph of an AC-based wDFA representing a set of strings $\mathcal{S} = \{he, she, his, hers\}$. The vertices and edges represent state and state transitions respectively.

The execution of the machine is illustrated in Fig.9.3. The DFA matches the input string $P = \{usherst \cdots\}$ against the string set $\mathcal{S} = \{he, she, his, hers\}$. For this purpose, each symbol of the input string is applied successively as an input symbol ${P^w}_i$ to the DFA. In one operating cycle, the next state and the output (accepting state indicator) are looked-up considering the current input symbol ${P^w}_i$ and the current state q_c. During each cycle, the next input symbol and the next state are captured and

Table 9.1: State transitions of the classic AC-based wDFA ($w = 1$) representing a set of strings $S = \{he, she, his, hers\}$.

$\Delta = (q_c, P^w{}_i, q_n)$			Transition Types	
q_c	$P^w{}_i$	q_n		
0	h	1	Δ_1	Δ_C
	s	3	Δ_1	Δ_U
	others	0	Δ_0	Δ_U
1	e	2	Δ_2	Δ_A, Δ_C
	i	6	Δ_2	Δ_U
	h	1	Δ_1	Δ_U
	s	3	Δ_1	Δ_U
	others	0	Δ_0	Δ_U
3,7,9	h	4	Δ_2	$(3, h, 4) \in \Delta_C$, others $\in \Delta_U$
	s	3	Δ_1	Δ_U
	others	0	Δ_0	Δ_U
2,5	r	8	Δ_3	Δ_U
	h	1	Δ_1	Δ_U
	s	3	Δ_1	$(2, s, 3) \in \Delta_C$, $(5, s, 3) \in \Delta_U$
	others	0	Δ_0	Δ_U
6	h	1	Δ_1	Δ_U
	s	7	Δ_3	Δ_A, Δ_C
	others	0	Δ_0	Δ_U
4	e	5	Δ_3	Δ_A, Δ_C
	i	6	Δ_2	Δ_U
	h	1	Δ_1	Δ_U
	s	3	Δ_1	Δ_U
	others	0	Δ_0	Δ_U
8	h	1	Δ_1	Δ_U
	s	9	Δ_4	Δ_A, Δ_C
	others	0	Δ_0	Δ_U

they are updated at the beginning of the next cycle. The exemplified machine starts the execution at time t_0 being at the initial state $q_c = 0$. Between the time interval $t_1 - t_0$ (the first operating cycle), the state transition rule $\Delta(0, u, 0)$ is executed and the current state remains state 0. Besides, the output also remains zero due to the fact that state 0 is not an accepting state. During this time interval, the next input symbol $\{s\}$ is also captured, which is performed by another machine that feeds the DFA with input symbols and the current input symbol is updated at time t_1. The following next cycles are executed similarly. Note that at times t_6 and t_4 the DFA reaches accepting states $\{5, 9\}$ and outputs logic 1.

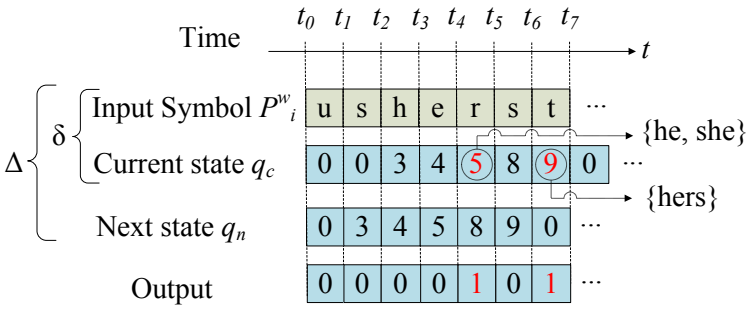


Figure 9.3: The execution of AC-based DFA matching the input string $\{usherst\cdots\}$ against the string set $\mathcal{S} = \{he, she, his, hers\}$. The red state numbers depict the visited accepting states.

9.1.1.2 Hardware Implementations of wDFA

The transition conditions of wDFA can be naively implemented into a RAM as a look-up table in the form of 2D matrix. Each row (or entry) stores the next state and the status of the accepting state flag which is the output. Each row is addressed by the associated transition condition δ as given in Fig. 9.4. The delay (or memory) can be implemented with a register, namely, State Register (SR). To naively implement a wDFA, $|Q| \cdot |\Sigma^w|$ rows, each of which is $(\lceil log_2|Q| \rceil + 1)$-bits long, are required. For example, DFA given in 9.1.1.1 requires 2560*5 = 12.8kbit RAM. After addressing the RAM with current transition condition, the next state q_n is loaded from the associated RAM entry and then it is stored in the state register as a current state q_c. After that, the cycle begins again. In this approach, all of the transitions are stored in a RAM. Hence, prohibitively large amount of memory may be needed to store large DFAs, especially for multi-byte automatons. For instance, Aho-Corasick based DFA storing SNORT v2.9 string set requires 50.61MByte RAM for $w = 1$ and 12.38GByte RAM for $w = 2$.

Another hardware implementation of wDFA is depicted in Fig.9.5. The state transition rules $\Delta = (q_c, P^w{}_i, q_n)$ and accept state flags are *completely* stored in a look-up table (LUT). This table can be realized with a RAM which is addressed by a CAM. In such implementation, the transition conditions $\delta = (q_c, P^w{}_i)$ and the associated next states q_n are stored in each entry of CAM and RAM, respectively. The values of accepting state flag ($m_f = mf_{LUT}$) can also be stored in RAM. Each state can be

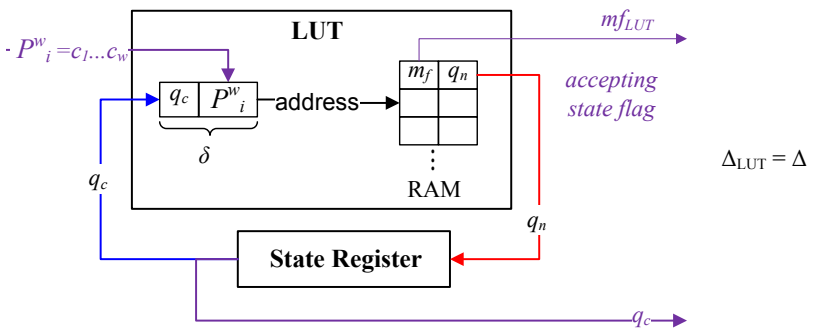


Figure 9.4: RAM-based naive implementation of wDFA.

coded with $|q| = \lceil log_2 |Q| \rceil$ bits. Therefore, to implement wDFA in this structure, we need $|\Delta| \cdot |q| \cdot w \cdot 8$ bits CAM and $|\Delta| \cdot (|q| + 1)$ bits RAM. For example, DFA given in 9.1.1.1 requires 20.48kbit CAM and 12.8kbit RAM, which means that *completely* storing all of the transitions Δ results in less efficiency in terms of memory usage. Instead, Δ_0 transitions, which are in large quantities (for example, $\Delta_0/\Delta = 99\%$ as in Table 9.1), can be inferred by the help of the match flag of CAM.

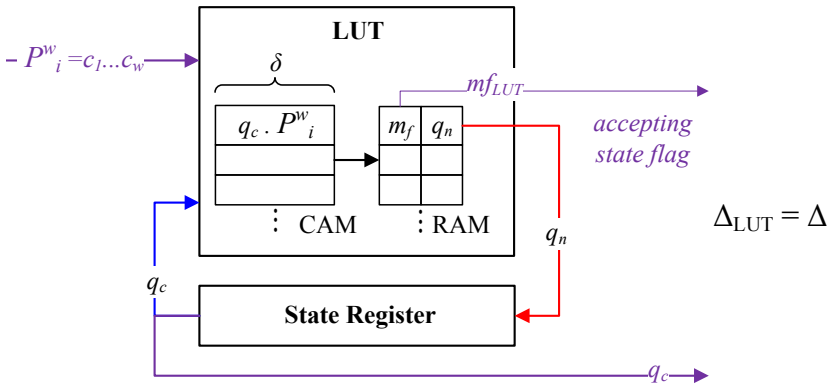


Figure 9.5: Typical hardware implementation of wDFA. LUT stores all of the transitions Δ. The transition conditions δ and the associated next states q_n (with the accepting state flags) are stored in a CAM and a RAM, respectively.

Large DFAs, for example string matching automatons, typically consist of large number of Δ_0 transitions. Therefore, instead of storing them explicitly in a LUT, *inferring* these transitions saves significant amount of memory resources. Inferring process can be performed by resetting the state register to initial state q_0 when CAM unmatches. In this architecture, LUT only sores $\Delta \setminus \Delta_0$ transitions. Further im-

provement can be done with inferring the value of accepting state flag, which can be realized with assigning a separate CAM to store the transitions to accepting next states Δ_A. This improved version is illustrated in Fig.9.6. This implementation necessitates $|\Delta \setminus \Delta_0| \cdot |q| \cdot w \cdot 8$ bits CAM and $|\Delta \setminus \Delta_0| \cdot |q|$ bits RAM. For example, DFA given in 9.1.1.1 requires 4 transitions $\Delta_A = \{(1, e, 2), (6, s, 7), (4, e, 5), (8, s, 9)\}$ to be stored directly into CAM_A and RAM_A. In addition, 22 transitions $\Delta \setminus (\Delta_0 \cup \Delta_A) = \{(0, h, 1), (0, s, 3), (1, i, 6) \cdots\}$ are stored into CAM_B and RAM_B. The remaining $\Delta_0 = 2534$ transitions are inferred. Therefore, 0.832kbit CAM and 0.104kbit RAM are necessary to implement DFA. It means that 24.62 and 123.08 times less CAM and RAM respectively are needed comparing to the previous implementation method illustrated in Fig.9.5.

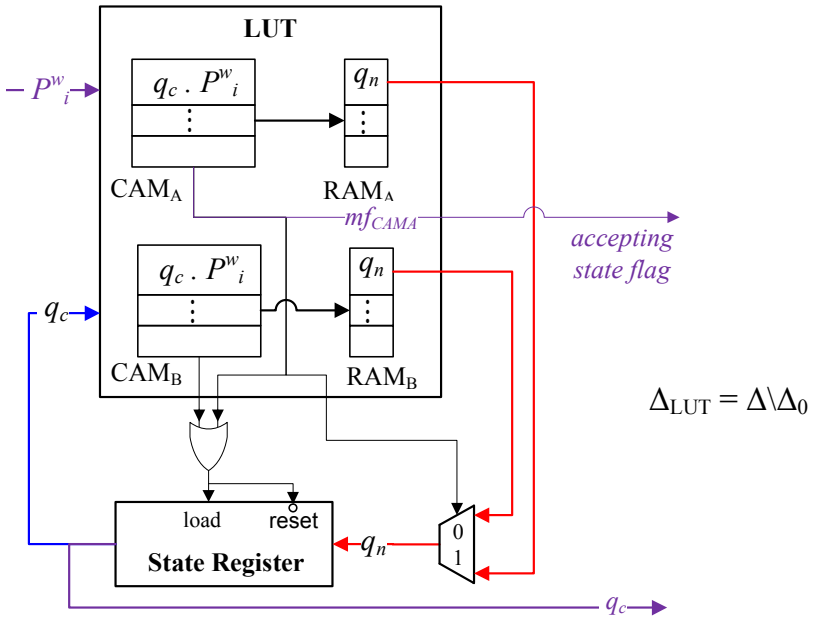


Figure 9.6: Improved hardware implementation of wDFA. CAM$_A$ stores the transitions to the accepting next states $\Delta_A = \{(q_c, P^w{}_i, q_n), q_n \in G\}$ and CAM$_B$ stores $\Delta \setminus (\Delta_0 \cup \Delta_A)$. Δ_0 transitions are inferred by the unmatch case of both CAMs. The accepting state flag is also inferred by the match case of CAM$_A$.

9.1.2 Bloom Filter based wDFA

For a typical wDFA, the number of all possible transitions is $|\Delta| = |\Sigma^w| \cdot |Q|$ and it is in line with the memory size. Therefore, for large DFAs, typical implementation

techniques may not be enough to realize the machine in terms of execution speed, memory size and cost. Hence, we need to optimize the resource usage of expensive hardware to increase feasibility and efficiency of the machine.

Bloom Filters are memory-efficient, probabilistic, multi-hashing data structures with controllable false positive probability and zero false negative probability. Due to non-zero false positives, a match result of the filter has to be verified by a block, namely, Verification Unit (VU). This unit can be realized with a low cost engine without any particular optimization; therefore the VU is expected to be a low speed and low cost engine. For instance, software based implementation techniques can be employed for this purpose.

Classical Bloom Filters [8] can be employed to compactly store some of the transition conditions of a DFA and we call this kind of machine as a Bloom Filter based Deterministic Finite Automaton (BFwDFA). In the literature, Dharmapurikar and Lockwood propose an Aho Corasick-based NFA, where state transition conditions are stored in BFs [16]. If there is a match result for the current transition condition δ, then this match can be verified by VU and the associated next state q_n can be achieved from the VU. Therefore, $\Delta \setminus \Delta_0$ transitions can be stored in a slow but low cost VU and the associated transition conditions $\delta \setminus \delta_0$ can be compactly represented by a high speed BF. Hence, similar to the improved hardware implementation of wDFA, Δ_0 transitions can be inferred by the unmatch case of the filter without any false results because of zero false negative probability of BF. Fig. 9.7 illustrates the architecture of the proposed BFwDFA and Algorithm 2 describes the execution of the machine. If the machine returns to the initial state q_0 most of the time, as in BF-based SMM executing under the low attack (or positive) rates, the disadvantages of the verification unit is diminished and we can take the advantage of the overall architecture in terms high rate of *state transitions per time*.

9.2 String Matching with Aho-Corasick Finite Automaton

In this section, we present the necessary background to make clear some basic concepts behind string matching procedures of Aho-Corasick Finite Automaton. For

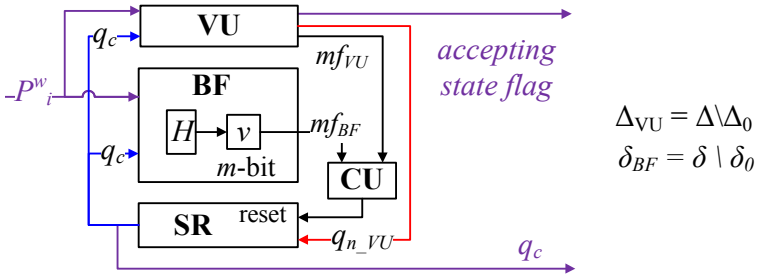


Figure 9.7: Structure of the BFwDFA.

more details, the original paper [8] can be referred. Moreover, lemmas and theorems related on the AC-based automatons are placed at the end of this section.

Automaton-based string matching approach, firstly, pre-processes the string set and builds an automaton accordingly. Then, during the query process, partial matches of the incoming strings are matched sequentially through state transitions. If the automaton reaches one of the accepting states, then the related string is *accepted* (or matched) by the machine. Entering an accepting state indicates that a non-empty set of strings are found in the input string. Input string is processed using a window of w bytes and advancing one byte at a time.

AC is the de-facto standard for multi-string matching engines, which matches all of the strings $\mathcal{S}$ at each operating cycle. The algorithm constructs an automaton-based string matching engine to process the input string by successively reading the input symbols, making appropriate state transitions and occasionally producing output. In typical implementations, each symbol is a single byte (8 bits). The construction of AC-NFA is derived from three functions: (1) a *goto* function g, (2) a *failure* function f, and (3) an *output* function. The function f can force the automaton to be multiple states. In these condition, the machine stops consuming input symbols, which decreases the scan rate. However, the equivalent deterministic version (AC-DFA), which consumes an input symbol for each state transition, can be generated by a *next move* function nm. Output function assigns some states as output states to report matching a subset of string set $\mathcal{S}$.

Construction of the goto transitions is described in Algorithm 3. The algorithm begins with adding an initial state 0 to the state transition graph and assigning return to zero

Algorithm 2 Control Unit(CU) function of basic BFwDFA

Initially: The machine is in the initial state q_0
For Each Queried Input Symbol ${P^w}_i$:
if BF matches **then** ▷ *Approximate* matching occurs
    wait VU
    if VU matches **then** ▷ True positive output for BF
        $q_n = q_{n\_VU}$
        if $q_n \in G$ **then**
            set accepting state flag ▷ A non-empty set of string is matched
        else
            reset accepting state flag ▷ No string is matched
        end if
    else
        $q_n = q_0$ ▷ Reset SR to state q_0
    end if
else
    $q_n = q_0$ ▷ Reset SR to state q_0
end if

transitions for all possible symbols from state 0. Then each symbol of each string from $\mathcal{S}$ is added by generating a new state transition condition and inserting a new state labeled with next successive numbers. If there is a common transition, then we skip them. For example, the Fig.9.8 illustrates the construction of goto function with $\mathcal{S} = \{he, she, his, hers\}$. For each string $S_i \in \mathcal{S}$, the current state q is assigned as accepting state and it is related with the current string S_i at the end of the process.

The construction of the failure transitions f is described in Algorithm 4. The algorithm begins with storing all of the depth-1 states Q_1 to a $queue$. The main $while$ loop stores the next depth states to the queue to process later. The algorithm also adds new strings to the accepting states.

The graph in Fig. Fig.9.9 exemplifies all of the goto g and failure f transitions of an AC-NFA machine storing the string set $\mathcal{S} = \{he, she, his, hers\}$. For example, the goto transition edge labeled s from state 8 to 9 indicates that $g(8, s) = 9$. The

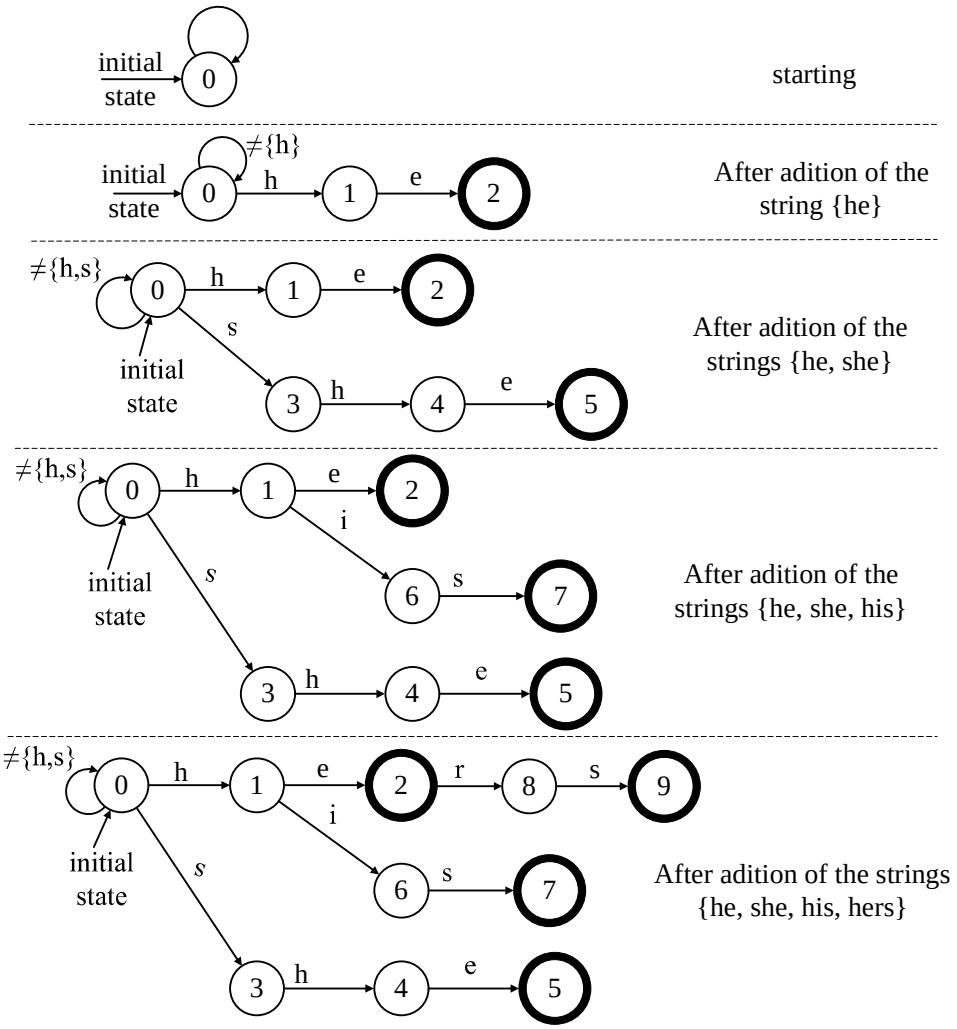


Figure 9.8: String-by-string construction of goto transitions g with string set $\mathcal{S} = \{he, she, his, hers\}$

absence of a goto transition edge represents $fail$ case. At state 8, other symbols than s $g(8, \neq s)$ results in fail. Note that the initial state 0 is defined in such a way that it has not any failure transition. If the goto function reports a fail case for a pair consisting of a current state q_c and current input symbol ${P^w}_i$, then the next state q_n is achieved from the failure transition. For example, AC machine returns directly to initial state 0 for all fail cases of goto function from states 1,2,3,6,8. Accepting states indicate that non-empty set of strings has been found. In Fig.9.9 the strings associated with the accepting states (depicted as bold vertices) are also indicated. Note that each accepting state label can also be used to represent the matched strings.

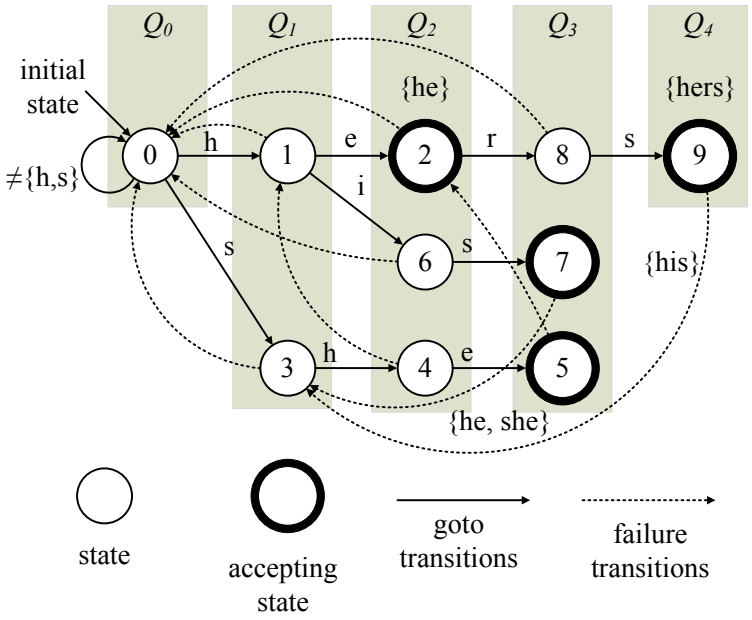


Figure 9.9: State transition graph of an AC-NFA machine storing the string set $\mathcal{S} = \{he, she, his, hers\}$

After the construction of the AC automaton, then it can be applied as string machine machine. The graph in Fig.9.10 illustrates the behavior of the AC NFA representing $S = \{he, she, his, hers\}$. Each symbol of the input string $P = usherst \cdots$ is applied to machine one by one. The appropriate next state q_n is calculated by the goto function, i.e. $q_n = g(q_c, Pw)$. When the goto function fails for an q_c and $P^w{}_i$ pair, failure transition takes the machine to a certain state where the goto function is reexecuted. It is possible that there are multiple failure transitions back to back and it is possible that the machine returns to the initial state afer such chain of failure transitions.

For example, at state 9 the goto function fails with symbol t. Thus, the state 3 of the failure function is tried, which is also fail, i.e. $g(f(9) = 3, t) = fail$. Therefore, $g(f(3) = 0, t)$ is tried which turn backs the machine to the initial state 0. Executing the goto function reqiures one operation cycle. It means that, AC NFA can consume more clock cycles to achieve the next state, which decreases the scan rate of the machine.

The input stream of bytes P is inspected symbol by symbol by AC automaton. If a transition condition $\delta = (q_c, P^w{}_i)$ is fulfilled, then the DFA applies the associated

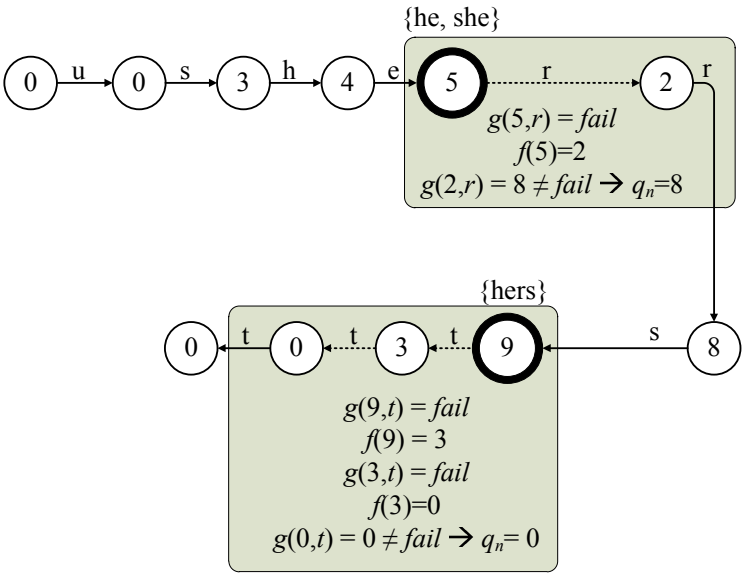


Figure 9.10: The behavior of the AC-NFA matching the input string $P = \{usherst\}$ against the string set $S = \{he, she, his, hers\}$.

transition rule $\Delta : (q_c, P^w{}_i, q_n)$ to enter the next state q_n. Following that, the input string is shifted one symbol to get the next input symbol.

For an AC-DFA, the number of state transitions required to process an input string S_n is independent of the string set cardinality $|\mathcal{S}|$. Therefore, AC-DFA is not vulnerable to burst or high rate queries (or attacks), making it very attractive to security systems. However, implementing a large AC-DFA representing a large string set into hardware decreases the maximum clock frequency. Even, implementing by software, which sustains high scan rate, is not possible; because, large number of state transition rules cannot be stored entirely in a cache memory. Although AC-NFA machines generate smaller transition tables, they are generally inefficient due to limited parallelism capability of hardware.

Table 9.1 exemplifies all of the state transitions of the AC-based DFA representing a set of strings $\mathcal{S} = \{he, she, his, hers\}$[8]. This machine consumes one-byte at a time, which means that $w = 1$. Transition types are listed in Table 9.1 and they are also illustrated in Fig.9.11.

Let define symbol σ^w to indicate the employed alphabet of a set of strings $\mathcal{S}$, where

$\sigma^w \in \Sigma^w$. Although the alphabet size is $|\Sigma^{w=1}| = 256$, only 5 symbols are employed by the set, i.e. $\sigma^{w=1} = \{h, s, e, i, r\}$. The total number of state transitions is $|\Delta| = 2560$. Most of them are to initial state q_0 and they are counted by $|\Delta_0| = 2534$. 15 transitions are to states of depth-1 $Q_1 = \{q_1, q_3\}$. 5 of the remaining 11 transitions are consecutive $\Delta_C \cap \Delta_{i\geq 2} = \{(3, h, 4), (4, s, 5), (6, s, 7) \cdots\}$ and other 6 transitions are unconsecutive $\Delta_U \cap \Delta_{i\geq 2} = \{(7, h, 4), (0, s, 3), (4, i, 6) \cdots\}$. It can be observed that many transitions occur to states whose depths are low. For example, 2549 of the overall 2560 transitions are to states of depth-0 and depth-1. The number of other transitions is remarkably small.

Main causes of these effects are listed as follows:

- Through matching a set of strings, almost all states, say q_c, expect small number of symbols (typically one symbol) to enter the next state q_n, where $depth(q_c) \leq depth(q_n)$ (please refer to Fig.9.11). For example, only one symbol (r,e or s) is needed to move the machine from depth-2 states 2,4,6 to depth-3 states 8,5,7 as indicated in Fig.9.9 and Table 9.1.

- Almost all other symbols make the machine enter zero state q_0 due to failure transitions. For instance, from all states, the machine returns back to the initial state 0 with ≥ 252 symbols (Note that $|\Sigma^{w=1}| = 256$).

- At a state, approximately $|Q_1|$ number of symbols moves the machine to depth-1 states Q_1 in order to match initial symbols of some new incoming strings. For example, states 0,1,2,4,5,8 need $|Q_1| = 2$ number of symbols (h or s) to match any strings from S (12 transitions). Similarly, states 3,7,9 and states 6,8 require symbols s and h to match strings $\{she\}$, $\{he, his, hers\}$ respectively (5 transitions). All of these 17 transitions move the machine to one of the dept-1 states $Q_1 = \{1, 3\}$.

- At a state, relatively small number of remaining symbols is needed to match some new strings due to *failure* transitions as described in [8]. For example, the machine moves to state 4 from states 7 and 9 with symbol h. Please refer to Fig.9.9.

- *goto* function [8] mainly generates consecutive transitions. For example, {(0,h,1),

(1,e,2), (8,s,9), (6,s,7), (3,h,4), (4,e,5)} transitions are consecutive and {(0,s,3), (1,i,6), (2,r,8), (4,e,5)} are unconsecutive goto transitions (Fig.9.9). The number of consecutive goto transitions may be increased as described in Theorem-2 (Section9.2.2).

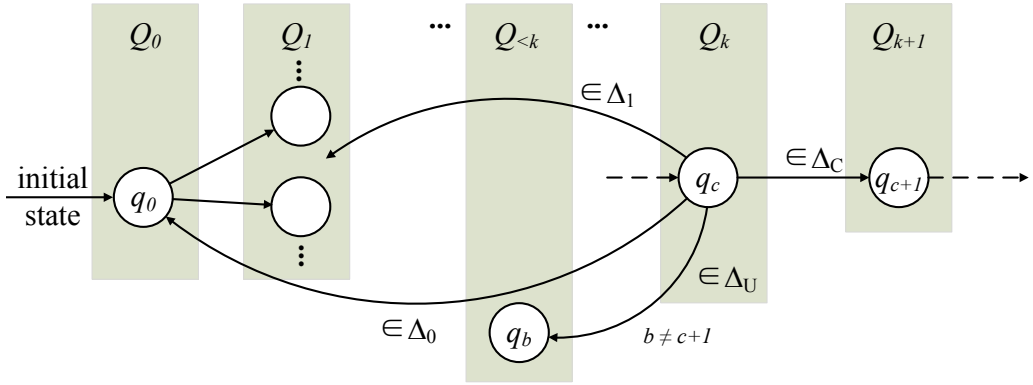


Figure 9.11: Illustration of transition types of a state q_c by a directed graph, whose vertices and edges represent states and state transitions respectively.

These observations are justified with AC-based wDFA implementations storing the string set of SNORT v2.9 as given in Section9.3. Moreover, they can be employed to map state transitions into fast hardware blocks as explain below:

- Instead of exactly *storing* Δ_0 transitions, they can be *inferred*. For example, if $\Delta \setminus \Delta_0$ transitions are stored in a look-up table, then any unmatch case of this table infers Δ_0 transitions. For a typical AC-based automaton, there are lots of Δ_0 transitions and therefore significantly large amount of memory can be saved by inferring them.

- The consecutive transitions Δ_C can also be inferred. We can approximately store them in a Bloom filter and in the case of any match, the next state can be calculated easily by incrementing the current state value. Due to the algorithm behind the *goto* function, we expect many consecutive transitions, hence inferring them enables us saving memory resources.

- Consider two transitions $(q_c, P^w{}_i, q_n)$ and $(q_c\prime, P^w{}_i\prime, q_n\prime)$. If $q_n = q_n\prime \in Q_1$, then $P^w{}_i = P^w{}_i\prime$. It means that, independent of the current state q_c, all of the

depth-1 states are one-to-one related with their associated input symbols $P^w{}_i$, which is proven by Theorem-1 (9.2.2). Hence, Δ_1 transitions can be inferred by a look-up table storing *only* $P^w{}_i$ and q_n pairs. For example, for all Δ_1 transitions, independent of the current state, the input symbols h and s moves the machine to states 1 and 3, respectively as depicted in Fig. 9.2 and Table 9.1. Note that the states 1 and 3 are members of Q_1.

9.2.1 Aho-Corasick Based Multi-Byte DFA

Instead of a single byte, w bytes can be consumed (or advanced) at a time, which enables us to achieve w times scan rate. This kind of machine can be constructed with w number of wDFAs, each of which concurrently consumes w-byte long input symbol $P^w{}_i$ at a time and requires O(1) computations per symbol.

The proposed machine is illustrated for $w = 2$ in Fig.9.12. There are two AC-2DFA storing the same set. Each DFA consumes $w = 2$-byte concurrently at a time and their search window is shifted by one-byte with respect to each other. After processing the current substrings on the current search windows, the overall search window is shifted (or advanced) by $w = 2$ bytes, which results in 2 times speedup in the scan rate.

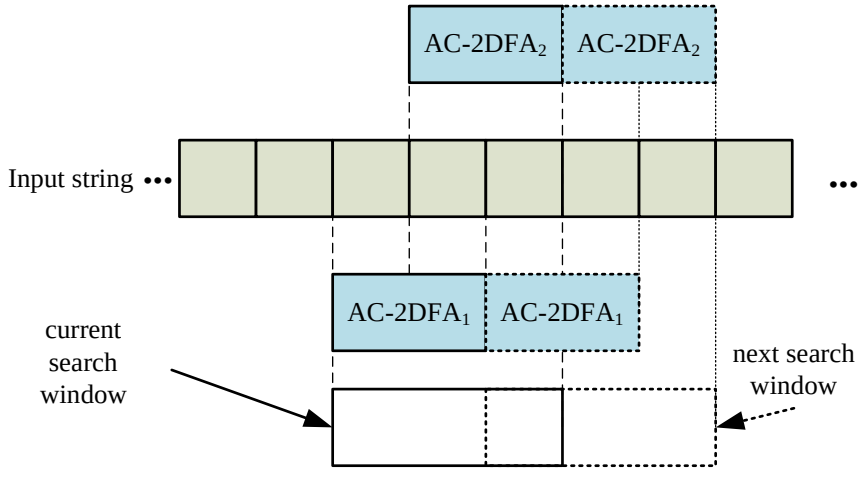


Figure 9.12: Illustration of AC-wDFA based SMM for $w = 2$.

To demonstrate the execution of the machine, we construct a simple SMM storing

the set $\mathcal{S} = \{he, she, his, hers\}$. The state transition graph is given in Fig. 9.13, which depicts only the *goto* transitions for simplicity. In addition, all of the transition rules are compactly represented in Table 9.2. The accepting states $G = \{1, 3, 5, 6\}$ represent all strings of the set $\mathcal{S}$. States are also depicted into groups considering their depths, namely, zero depth state $Q_0 = \{0\}$, depth-1 states $Q_1 = \{1, 2, 4\}$ and depth-2 states $Q_2 = \{3, 5, 6\}$. At a state, more than one symbol can match to move the machine into two different next states. Under the case of multiple matches of transition conditions, the next state of the longest input symbol is considered.

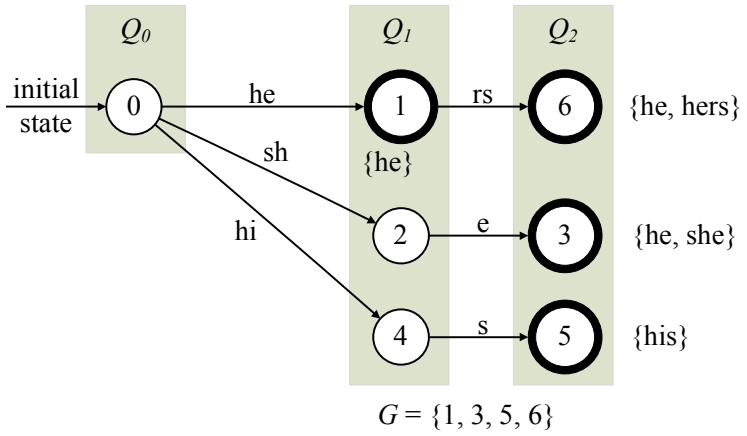


Figure 9.13: State transition graph for the AC-2DFA based SMM storing the string set $\mathcal{S} = \{he, she, his, hers\}$.

Table 9.2: Compact representation of state transition rules of the classic AC-2DFA machine storing the set of strings $\mathcal{S} = \{he, she, his, hers\}$. The priorities of the rules reduce towards bottom row.

$\Delta = (q_c, {P^w}_i, q_n)$			Transition Types			
q_c	${P^w}_i$	q_n	Δ_0	Δ_1	$\Delta_{i\geq 2}$	
					$\Delta_C \cap \Delta_{i\geq 2}$	$\Delta_U \cap \Delta_{i\geq 2}$
1	rs	6				✓
2	e	3			✓	
4	s	5			✓	
any state	he	1		✓		
	sh	2		✓		
	hi	4		✓		
any state	any symbol	0	✓			

For the input string $P = \{usherst \cdots\}$, the appropriate state transitions and the associated accepted strings are given in Fig.9.14. With current input symbol ${P^w}_i =$

$\{he\}$ and current state $q_c = 0$, AC-2DFA$_1$ moves to state 1 and accepts the string $\{he\}$. After that, this machine accepts the strings $\{he, hers\}$ through visiting state 6. Similarly, the strings $\{he, she\}$ are matched by the automaton AC-2DFA$_2$ at state 3.

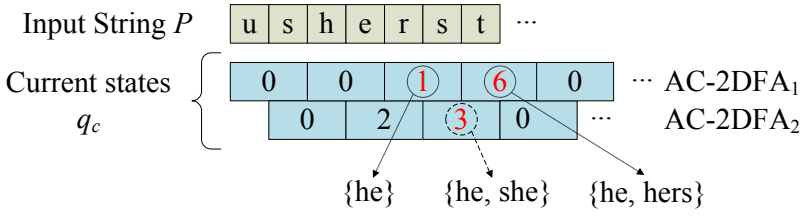


Figure 9.14: The execution of the AC-wDFA based SMM matching ($w = 2$) the input string $P = \{usherst \cdots\}$ against the string set $\mathcal{S} = \{he, she, his, hers\}$. The red state numbers depict the visited accept states.

9.2.2 Lemmas and Theorems on AC-based Automaton

For an AC machine, Algorithm 4 constructs a deterministic finite automaton from the goto function g (Algorithm 2) and failure function f (Algorithm 3), as given in [8]. The following theorems, lemmas and their related proofs are based on these algorithms.

Lemma 1: For all of the goto functions $g(q_c, P^w{}_i) = q_n$, the inequality $depth(q_c) \leq depth(q_n)$ is valid.

Proof: The inequality $depth(q_c) \leq depth(q_n)$ is valid for all of the goto functions $g(q_c, P^w{}_i) = q_n \neq fail$ due to incremental state assignment, $newstate = newstate + 1$ except $g(0, P^w{}_i)$, as defined in Algorithm 2. All of the goto transitions such that $g(0, P^w{}_i) = fail$ are assigned to initial state 0 as next state, therefore the depth remains same.

Lemma 2: Every transitions to next state q_n of depth ≤ 1 is due to (1) goto function $g(q_c = 0, P^w{}_i)$ and (2) failure function $f(q_c) = 0$.

Proof: The proof proceeds by induction on the depth of current state q_c. There is only one current state q_0 on the depth of 0. By Algorithm 2, for all $P^w{}_i$ such that $g(0, P^w{}_i)$ is $fail$, then the goto function defines the next state to q_0, so the transition is to depth

of 0. Due to the definition of $depth$ function, all other transitions ($g(0, P^w{}_i)$ is not $fail$) are to states of depth 1. For the q_c of depth $1 \leq$, due to the Lemma 1, there is not any goto transition to q_n of depth ≤ 1. Only the transitions due to the failure functions $f(q_c) = 0$ is to states of depth $1 \leq$.

Theorem 1: When the AC-DFA is in some current state q_c with a specific input symbol $P^w{}_i$, there is only one next state q_n at depth ≤ 1.

Proof: Every transitions to next state q_n of depth ≤ 1 is due to (1) goto function $g(0, P^w{}_i)$ and (2) failure function $f(s) = 0$ (Lemma 2). For each input symbol $P^w{}_i$, $g(0, P^w{}_i)$ values are assigned to a deterministic transition, i.e. $\delta(0, P^w{}_i) \longleftarrow g(0, P^w{}_i)$. Every failure functions that result in zero-value for some state s is assigned to deterministic transitions, i.e. $\delta(0, P^w{}_i) \leftarrow \delta(f(s) = 0, P^w{}_i)$. Because all of the transitions to next states at depth ≤ 1 is through the deterministic $\delta(0, P^w{}_i)$ transitions, the theorem is true.

Theorem 2: If the construction of the goto function g (Algorithm 2) begins with the longer strings, then the number of successive transitions such that $q_n = q_c + 1$ is maximum.

Proof: The proof proceeds by considering all of the relationships between two strings. Suppose that $u = u_1, u_2, \ldots, u_i$ and $v = v_1, v_2, \ldots, v_j$ are members of the string set S.

- Assume that u is a proper prefix of v, i.e. $u_1 = v_1, u_1 = v_1, \ldots, u_i = v_i$, and $j > i$. The longest string v can be processed (1) before or (2) after processing of u. In the first case (1), string v creates $j - 1$ successive transitions due to the inner for-loop of Algorithm 2. Following this, during the processing of u, i symbols (prefix) are ignored by the inner while-loop and no more transition is generated. At the end, $(j - 1)$ successive transitions are generated. (2) If u is processed before v, then $(i-1)+(j-i-1)$ successive transitions are generated sequentially. Therefore, we get one more successive transitions in the first case (1). That is because of having transition from v_i to v_{i+1} in a successive state assignment.

- Assume that u and v shares a prefix with l symbols with possibility of $l = 0$.

Then, $i + j - l - 2$ successive transitions occur, which is independent of the sequence of string processing.

- Because Algorithm 2 ignores sharing of suffix or infix (it is based on longest prefix matching), they do not effect the number of successive transitions.

Suppose that for a given set of string S, a number of η strings are proper prefix of some other strings with possibility of $\eta = 0$. Processing the longer strings first initiates processing some strings before their η proper prefixes and this enables us to achieve η more successive transitions than the case of processing the shorter strings (proper prefixes) firstly. Therefore, the theorem is true.

Theorem 3: The set of next states q_n of depth ≤ 1 is a subset of next states $q_n = \delta(0, P^w{}_i)$

Proof: Because all of the transitions to next states at depth ≤ 1 is through the deterministic $\delta(0, P^w{}_i)$ transitions, the theorem is true (Please refer to the proof of Theorem 1).

9.3 Mapping SNORT String Set into an AC-wDFA

In this Section, we present the construction of AC-wDFAs. w is a variable which determines how many bytes are consumed at a time by a wDFA as defined in Section 9.1.1. In this study we take w as $w = \{1, 2, 3, \cdots, 20\}$ for the signature string set of SNORT v2.9. For this purpose, a program is designed in Microsoft Visual C++ environment that takes a set of string $\mathcal{S}$ and w as main arguments and generates all of the $\Delta \setminus \Delta_0$ transitions which can be used to construct the machine (Please refer to Fig. 9.15). Some information, such as $|Q|$, $|Q_1|$, $|\sigma_w|$, $|\Delta_0|$, $|\Delta_1|$,$|\Delta_{i\geq 2}|$,$|\Delta_C \cap \Delta_{i\geq 2}|$, $|\Delta_U \cap \Delta_{i\geq 2}| \cdots$ are also reported. These outcomes are employed to support the observations given in Section 9.2 and calculate the memory efficiencies of the proposed SMMs summarized in Section 10.2.

The execution of the program is exemplified in Fig. 9.16 by an AC-wDFA ($w = 1$) storing the string set $\mathcal{S} = \{he, she, his, hers\}$.

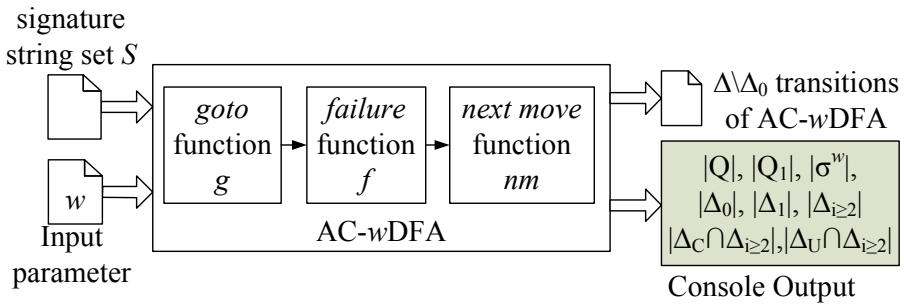


Figure 9.15: Mapping SNORT String Set into an AC-wDFA.

In the naive approach as shown in Figure 9.1, the transitions can be stored as 2D array where each combination of q_c and ${P^w}_i$ is related with q_n. However, the needed memory to handle the transitions of AC-DFA is in line with the size of the symbol alphabet $|\Sigma^w|$, therefore, for large w values, the execution of program is not possible for Microsoft Visual C++ environment, which limits the usage of memory upto 4G bytes. To cope with this problem, the transitions are stored in an array of linked lists as illustrated in Fig.9.17. In this data structure, each array cell depicts the current state q_c and stores the associated ${P^w}_i$ and q_n pairs as a linked list. Furthermore, the inferred Δ_0 transitions are not stored in the memory during the execution of the program.

Table 9.3 lists the alphabet and state parameters of the generated automatons for SNORT v2.9. On one hand, the alphabet size $|\Sigma^w|$ exponentially increases with w due to the fact that $|\Sigma^w| = 2^{8 \cdot w}$. On the other hand, the employed alphabet size $|\sigma^w|$ by the string set does not grow dramatically. With increasing w, the universal set Σ^w expands exponentially, as a result, the employment probability of the alphabet symbols (by the automaton) increases until $w = 4$. Because of sharing many elements of $|\sigma^w|$ among string set members, there is a local minimum at $w = 4$ as shown in Fig.9.18. For $w > 4$, this effect decreases, therefore, $|\sigma^w|$ decreases. With increasing w, the total number of states $|Q|$ decreases and the number of depth-1 states $|Q_1|$ increases as shown in Fig.9.19 and Fig.9.20, respectively.

The set cardinalities of the transitions are given in Table 9.4 and significant percentages among them are shown in Fig.9.21. For $w \geq 2$, Δ_0 and Δ_1 transitions cover almost all of the transitions, i.e. $|\Delta| \cong |\Delta_0| + |\Delta_1|$. As explained before, instead

Table 9.3: Alphabet and state parameters of the generated w-byte AC machines, each of which stores the signature strings of Snort v2.9.

w	$\lvert\Sigma^w\rvert$	$\lvert\sigma^w\rvert$	$\lvert\sigma^w\rvert/\lvert\Sigma^w\rvert$	$\lvert Q\rvert$	$\lvert Q_1\rvert$	$\lvert Q_1\rvert/\lvert Q\rvert$
1	$2^8 = 256$	92	35.94%	92133	84	0.09%
2	$2^{16} = 65536$	4162	6.35%	47669	941	1.97%
3	$1.68 \cdot 10^{07}$	12000	0.07%	33131	1730	5.22%
4	$4.29 \cdot 10^{09}$	14900	0.00%	25227	1966	7.79%
5	$1.10 \cdot 10^{12}$	14834	0.00%	21109	2330	11.04%
6	$2.81 \cdot 10^{14}$	13347	0.00%	17809	2493	14.00%
7	$7.21 \cdot 10^{16}$	12639	0.00%	15700	2575	16.40%
8	$1.84 \cdot 10^{19}$	11786	0.00%	14030	2748	19.59%
9	$4.72 \cdot 10^{21}$	10728	0.00%	12485	2806	22.47%
10	$1.21 \cdot 10^{24}$	10257	0.00%	11648	2863	24.58%

of exactly storing Δ_0 and Δ_1 transitions, they can be inferred, which saves significant amount of memory. Similarly, Δ_C transitions can be inferred by a Bloom filter. Verification of this filter can be performed easily by a lowly populated second filter, which is similar to the DBF-SMM architecture. However, Δ_U transitions can not be inferred by a Bloom filter, because these filters only stores the transition conditions δ, therefore, the associated next states must be stored exactly somewhere else. But a small portion of Δ_U can be stored in a look-up table in order to bypass the verification process for them. With increasing w, percentage of Δ_C transitions increases and percentage of Δ_U transitions decreases, which increase the memory efficiency of the proposed SMM.

Table 9.4: Set cardinalities of transitions of AC-wDFAs storing the signature strings of Snort v2.9.

w	$\lvert\Delta\rvert$	$\lvert\Delta_0\rvert$	$\lvert\Delta_1\rvert$	$\lvert\Delta_{i\geq 2}\rvert$	$\lvert\Delta_C \cap \Delta_{i\geq 2}\rvert$	$\lvert\Delta_U \cap \Delta_{i\geq 2}\rvert$
1	$2.36 \cdot 10^{07}$	$1.58 \cdot 10^{07}$	$6.31 \cdot 10^{06}$	1.425.724	89.026	1.336.698
2	$3.12 \cdot 10^{09}$	$3.08 \cdot 10^{09}$	$4.48 \cdot 10^{07}$	130.188	44.497	85.691
3	$5.56 \cdot 10^{11}$	$5.56 \cdot 10^{11}$	$5.73 \cdot 10^{07}$	42.540	29.924	12.616
4	$1.08 \cdot 10^{14}$	$1.08 \cdot 10^{14}$	$1.03 \cdot 10^{07}$	24.427	22.038	2.389
5	$2.32 \cdot 10^{16}$	$2.32 \cdot 10^{16}$	$9.46 \cdot 10^{06}$	19.101	17.896	1.205
6	$5.01 \cdot 10^{18}$	$5.01 \cdot 10^{18}$	$8.73 \cdot 10^{06}$	15.498	14.591	907
7	$1.13 \cdot 10^{21}$	$1.13 \cdot 10^{21}$	$8.07 \cdot 10^{06}$	13.253	12.484	769
8	$2.59 \cdot 10^{23}$	$2.59 \cdot 10^{23}$	$8.56 \cdot 10^{06}$	11.359	10.812	547
9	$5.90 \cdot 10^{25}$	$5.90 \cdot 10^{25}$	$6.48 \cdot 10^{06}$	9.717	9.264	453
10	$1.41 \cdot 10^{28}$	$1.41 \cdot 10^{28}$	$6.08 \cdot 10^{06}$	9.206	8.816	390

Algorithm 3 Construction of goto transitions [8]

Input: A set of strings $\mathcal{S} = \{S_1, S_2 \cdots S_n\}$

Output: Goto function g and a partially computed output function $output$

Note: The function $enter(S)$ inserts state transition graph a transition with symbol S.

Initially: $g(q_c, {P^w}_i) = fail$ for all $(q_c, {P^w}_i)$ pairs, $output(q_c)$ is empty, $newstate = 0$, and $i = 1$.

Begin

for $i \leq n$ **do**

    $enter(S_i)$, $i = i + 1$

end for

for all ${P^w}_i$ such that $g(0, {P^w}_i) = fail$ **do** ▷ Executed $|\Sigma^w|$ times !

    $g(0, {P^w}_i) = 0$

end for

End

function $enter(P_{w_1}, P_{w_2} \cdots P_{w_k})$

    $q = 0, j = 1$

    while $g(q, P_{w_j}) \neq fail$ **do** ▷ Skips the shared prefixes

        $q = g(q)$ ▷ The last state that we will add a new transition

        $j = j + 1$

    end while

    $p = j$

    for $p \leq k$ **do**

        $newstate = newstate + 1$ ▷ Generating a successive label

        $g(q, P_{w_p}) = newstate$ ▷ Generating a new goto transition

        $q = newstate$ ▷ Update the last state that we'll add a new trans.

    end for

    $output(q) = \{P_{w_1}, P_{w_2} \cdots P_{w_k}\}$ ▷ assigning the current string to $q \in G$,

end function

Algorithm 4 Construction of failure transitions [8]

1: Input: Goto transitions g and output function $output$ from Algorithm 3.
2: Output: Failure transitions f and final output function $output$
3: Initially: Queue $queue$ is empty.
4: **Begin**
5: **for** for each $P^w{}_i$ such that $g(0, P^w{}_i) = q_n \neq 0$ **do**
6: $queue = queue \cup \{q_n\}$ ▷ Store all depth-1 states Q_1 to the queue.
7: $f(q_n) = 0$ ▷ Assign all failure transitions of depth-1 state Q_1 to initial state 0.
8: **end for**
9: **while** $queue \neq empty$ **do**
10: $queue = queue - \{q_c\}$ ▷ Get the next state from the queue.
11: **for** each $P^w{}_i$ such that $g(q_c, P^w{}_i) = q_n \neq fail$ **do**
12: $queue = queue \cup \{q_n\}$ ▷ Add the next depth states to the queue.
13: $q = f(q_c)$
14: **while** $g(q, P^w{}_i) = fail$ **do**
15: $q = f(q)$
16: **end while**
17: $f(q_n) = g(q, P^w{}_i)$
18: $output(q_n) = output(q_n) \cup output(f(q_n))$
19: **end for**
20: **end while**
21: **End**

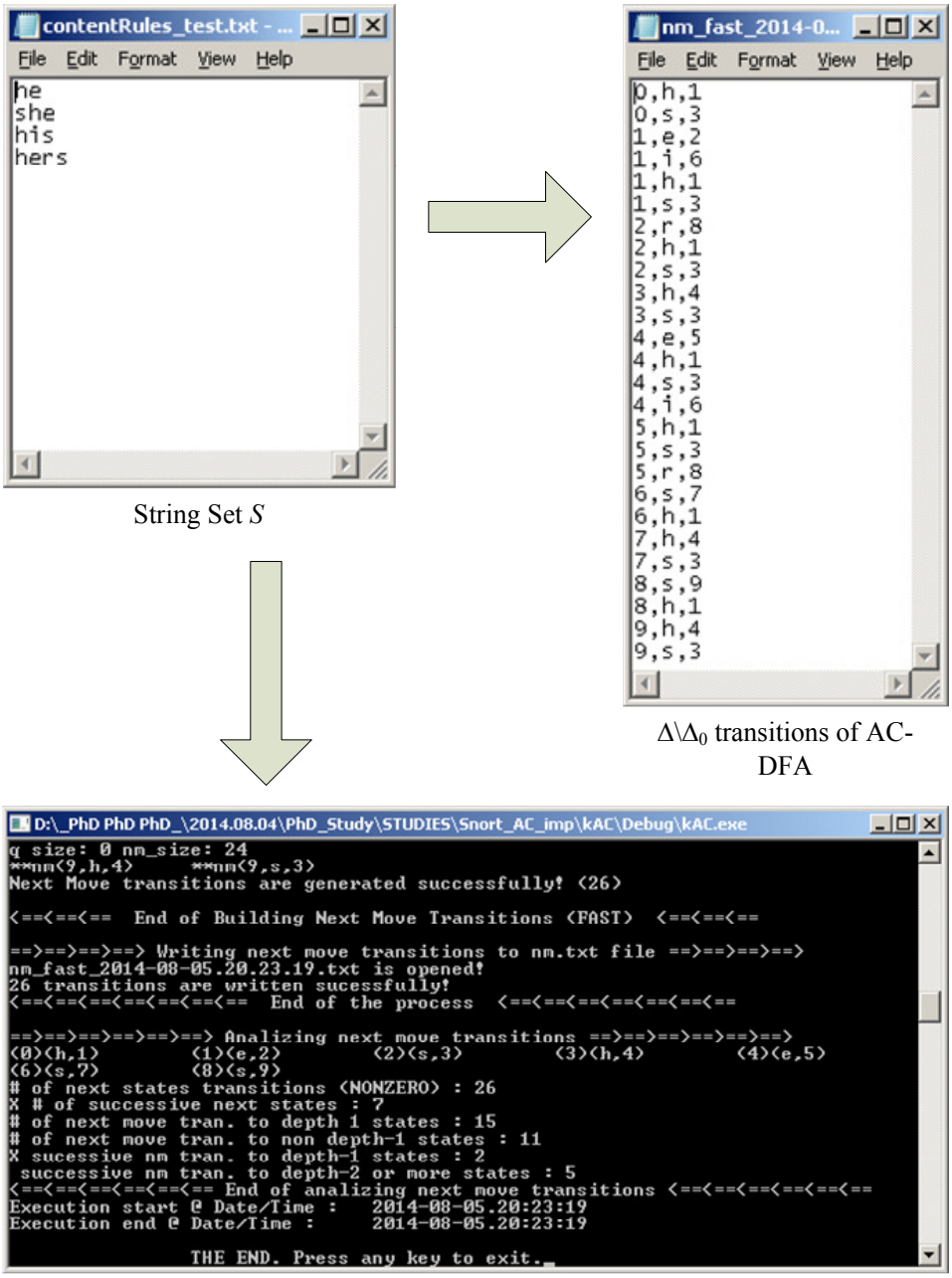


Figure 9.16: Mapping string set $\mathcal{S} = \{he, she, his, hers\}$ into an AC-wDFA.

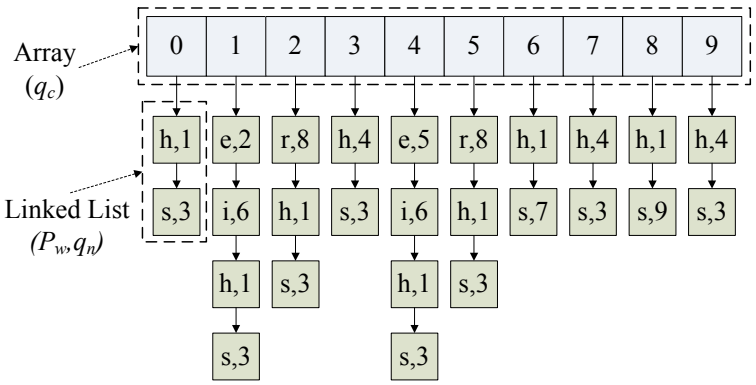


Figure 9.17: Array of linked list structure to represent the state transitions given in Table 9.1.

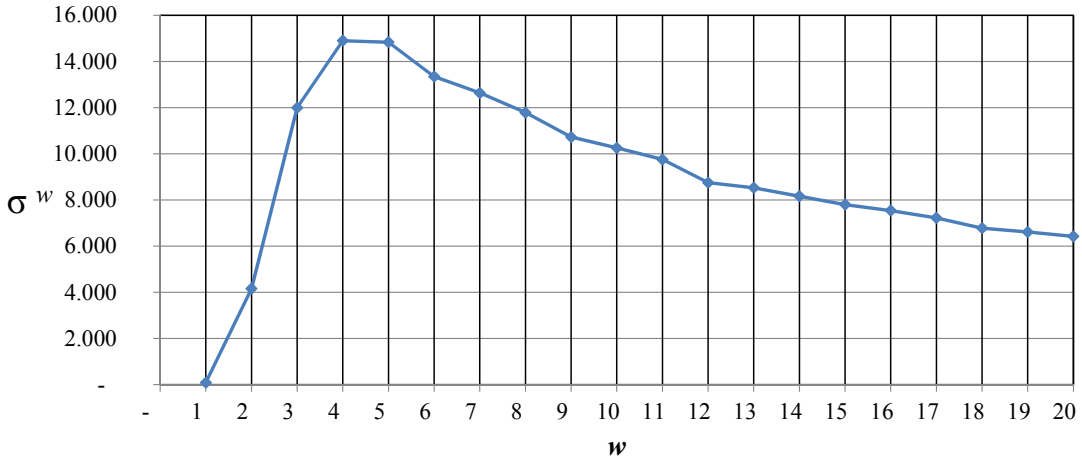


Figure 9.18: Employed alphabet size $|\sigma^w|$ as a function of w.

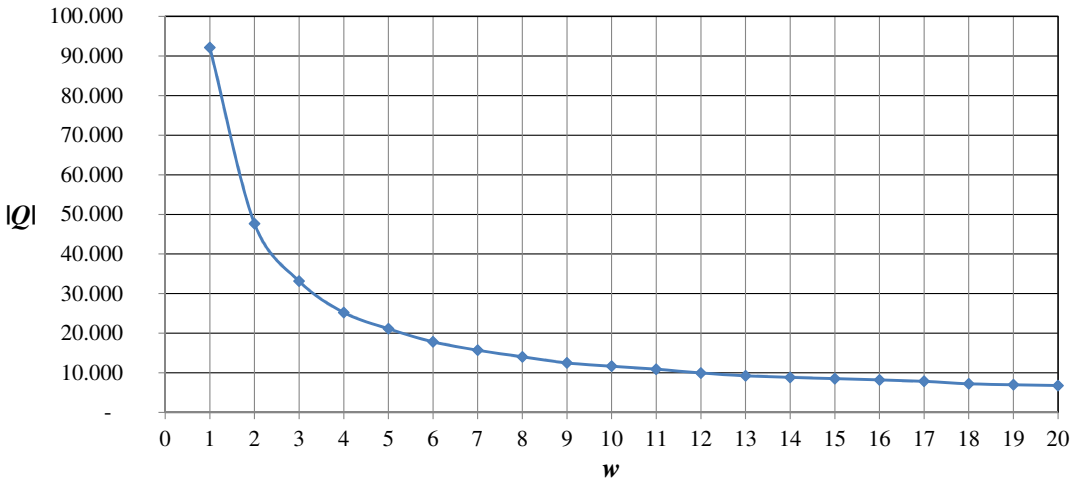


Figure 9.19: Total number of states $|Q|$ as a function of w.

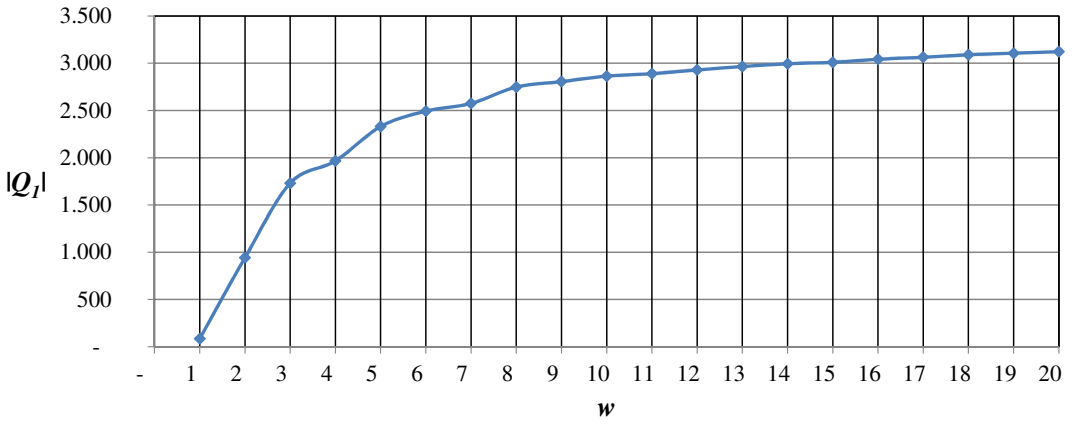


Figure 9.20: Total number of depth-1 states $|Q_1|$ as a function of w.

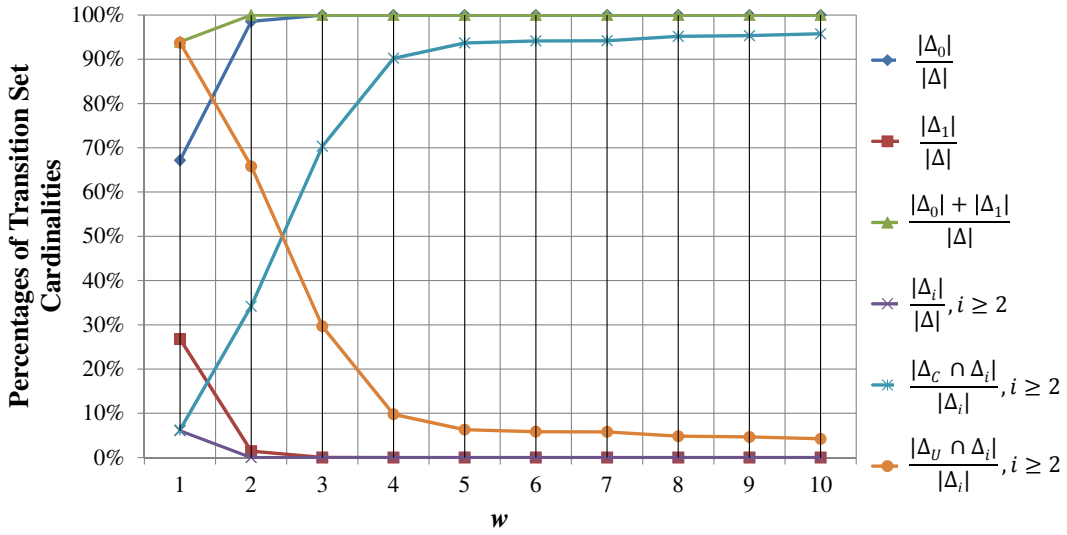


Figure 9.21: Percentage of transition set cardinalities as a function of w.

CHAPTER 10

VARIABLE SIZE STRING MATCHING WITH DOUBLE BLOOM FILTERS

10.1 String Matching Module with Bloom Filter based Aho-Corasick Automaton

Designing an automaton that matches (or accepts) a large set of string set is a challenging problem because high number of states and state transition rules emerge. These issues bring significant problems in realization of the automaton. These problems can be generalized into two types: (1) hardware resource usage (memory, logic gates...) and (2) execution speed.

The motivating idea behind of BF-based automaton is that storing the transitions in a hash based architecture may save the hardware resources significantly such as in [16]. They propose an NFA that is built upon AC[8] which consumes w-bytes at a time. The states and their associated symbols are stored in BFs. If any match occurs, then the next state transition is captured from the off-chip hash table, which slows down the scan rate under the burst queries.

In [37], BF-based heuristic is proposed in order to speedup the bitmap AC-based matching process. The slow automaton matching is bypassed if the initial BF does not produce a match for the input symbols mapped on the current search window. Thus, the architecture is only able to increase the scan rate under normal traffic conditions, where the number of true positive matches of BF is low.

10.1.1 Basic BFbAC-wDFA

As mentioned before, inferring the Δ_1 transitions can be performed for AC-based DFAs. Hence, different from the BFbDFA depicted in Fig. 9.7, we can add a look-up table in order to exactly store only the input symbols $P^w{}_i$ and their associated next states q_n (and ignoring the current state q_c). This architecture also bypasses the verification process for the Δ_1 transitions. Structure of the basic BFbAC-wDFA is illustrated in Fig. 10.1 and the Control Unit (CU) function is given in Algorithm 5.

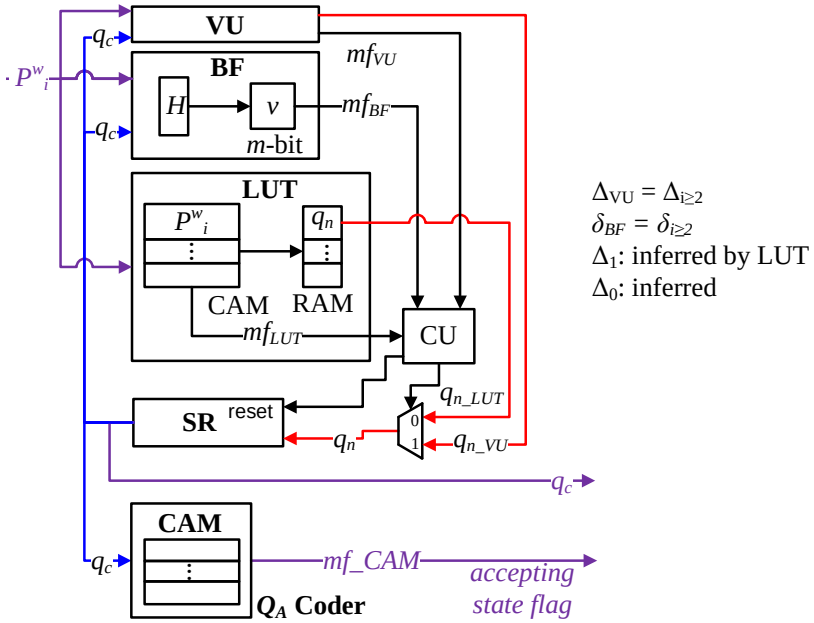


Figure 10.1: Structure of the basic BFbAC-wDFA.

Verification Unit *exactly* stores transitions $\Delta_{VU} = \Delta_{i\geq 2}$ and BF *approximately* stores transition conditions $\delta_{BF} = \delta_{i\geq 2}$. Δ_1 transitions are *inferred* by LUT when BF unmatches or BF matches falsely. Δ_0 transitions are also inferred by unmatch conditions of LUT with unmatch or falsely match conditions of BF. The accepting state flag can be generated with the match flag of a CAM storing accepting states Q_A. Note that, for 1-byte input symbol ($P_i^{w=1}$), LUT can be implemented only with a single RAM consisting of 256 locations. In this configuration, address bus of the RAM is connected to the input symbol $P_i^{w=1}$ and each RAM content stores the associated next state. Therefore, RAM always matches and instead of inferring the Δ_0 transitions they can also be exactly stored in RAM. For $w >> 1$, the number of required loca-

Algorithm 5 Control Unit(CU) function of basic BFbAC-wDFA.

1: Initially: The machine is in state q_0 (reset value of SR)
2: Condition: Main *if* functions are concurrent
3: For each queried input symbol ${P^w}_i$:
4: **if** BF matches **then** ▷ BF has higher priority than LUT
5: wait VU ▷ Due to verify the BF matches
6: **if** VU matches **then** ▷ True positive match for BF
7: $q_n = q_{n\_VU}$ ▷ Load the next state from VU
8: **else if** LUT matches **then** ▷ False positive match for BF
9: $q_n = q_{n\_LUT}$ ▷ Δ_1 transition occurs
10: **else** $q_n = q_0$ ▷ Return to initial state
11: **end if**
12: **else if** LUT matches **then**
13: $q_n = q_{n\_LUT}$ ▷ Δ_1 transition occurs
14: **else** $q_n = q_0$ ▷ Return to initial state
15: **end if**

tions becomes too large to be realized with a small RAM. For example, 8.6G bytes of RAM is needed for $w = 4$ by considering 2 bytes to code the states.

After the generation of AC-based wDFA from a set of string as described in Section 9.2, then we can determine the parameters of the proposed machine: The length of SR is $\lceil log_2|Q| \rceil$, where $|Q|$ is the cardinality of the set Q. To store $P_i^{w=1} \times q_n$ matrix for one byte input symbols, we need $2^8 = 256$ locations, each of which is $\lceil log_2|Q| \rceil$ bits in size. As a result, the overall RAM size is $256 \cdot \lceil log_2|Q| \rceil$ bits. For multiple-byte SMM ($w \geq 2$), $|Q_1| \cdot |{P^w}_i|$ bits CAM and $|Q_1| \cdot \lceil log_2|Q| \rceil$ bits RAM are needed to implement LUT. The BF parameters, i.e. the number of hash functions k and the size of the v-vector m, can be determined by the allowable false positive probability P_{fp} and the number of transition rules stored in, i.e. $|\Delta_{BF}|$.

After the construction of the proposed machine, it can be applied as a compact SMM. In each query process, both of the BF and LUT execute concurrently. For any transition condition $\delta(q_c, {P^w}_i)$, the BF can report a *positive* or *negative* match. Any match output of BF necessitates a query in order to verify this match. If a negative or a false

positive match occurs, then it means that there is not any real match in BF and the next state is loaded from the associated RAM location if LUT matches, i.e. $q_n = q_{n\_LUT}$. Otherwise, the state register is reset in order to move the automaton to the initial state q_0. If a true positive occurs, then the next state is loaded from VU, i.e. $q_n = q_{n\_VU}$. The accepting states Q_A are decoded concurrently with a CAM by activating the accepting state flag.

Fig. 10.2 exemplifies the memory contents of the proposed machine ($w = 1$) storing the set of strings $\mathcal{S} = \{he, she, his, hers\}$. All transitions of AC-$w$DFA storing this set is given in Table 9.1. The execution cycles of BFbAC-wDFA are demonstrated in Fig. 10.3, where the input string is $\{P = ushersy \cdots\}$. The next states of the transitions $(0, u, 0)$, $(0, s, 0)$, and $(9, y, 0)$ are loaded from the RAM. For these transitions, BF is not expected to match any false positive output due to low false positive rate. However, $(3, h, 4)$, $(4, e, 5)$, $(5, r, 8)$, and $(8, s, 9)$ transitions are matched by the BF and after the verification of each transition conditions, the associated next states are loaded from the VU.

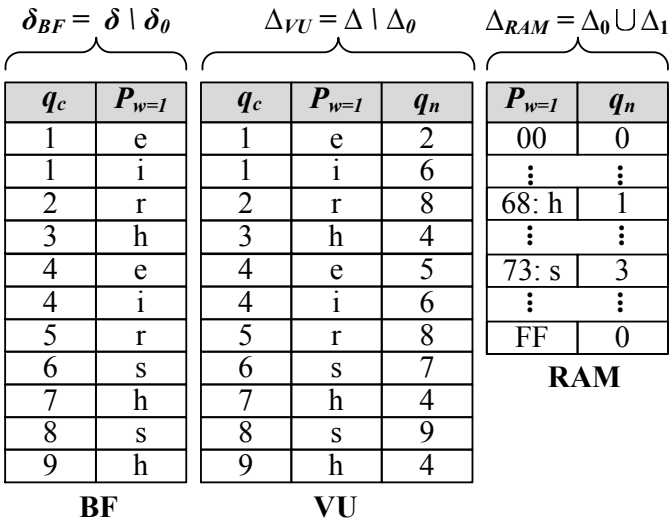


Figure 10.2: The RAM and BF content of the proposed machine storing the string set $S = \{he, she, his, hers\}$

The memory efficiency of BFbAC-wDFA is highly related on the number of inferred transitions, i.e. $|\Delta_0 \cup \Delta_1|$. Besides, the scan rate of the system depends on the number of slow VU queries, which is determined by the true and false positive rates of BF.

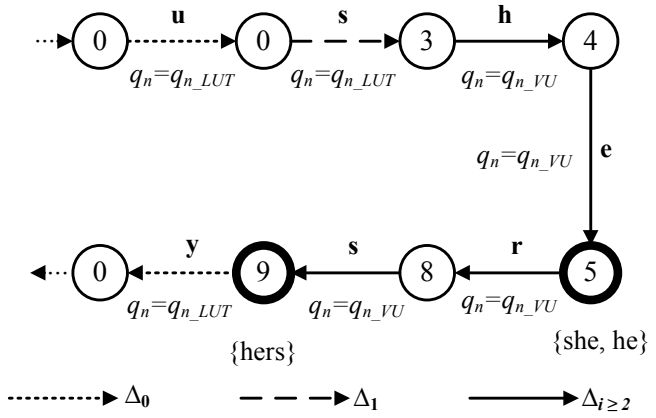


Figure 10.3: The execution of the proposed machine matching the input string $\{P = ushersy \cdots\}$ against the string set $\mathcal{S} = \{he, she, his, hers\}$. The machine begins consuming each byte on the left side of the input string.

True positive rate depends on the input symbol characteristics and BF design parameters determine the false positive rate. If a high number of $\Delta_0 \cup \Delta_1$ transitions occur, then a low number of slow verifications occur and the average scan rate increases. Otherwise, the machine slows down by spending more time in order to verify the current transition conditions and, as a result, the scan rate decreases. In other words, if the input strings consist of a low number of positive strings ($P_{\mathcal{S}} \cong 0$), then significant amount of the transitions are performed quickly by the fast RAM. However, if the input strings consist a large number of strings ($P_{\mathcal{S}} \cong 1$) from the set $\mathcal{S}$, then slow verification process is invoked frequently, which significantly decreases the scan rate of SMM.

10.1.2 Multiple BFbAC-wDFA

The scan rate of BFbAC-wDFA strongly depends on the input traffic characteristic. Under frequent positive events ($P_{\mathcal{S}} \cong 1$), a high number of references to the VU occur, which slows down the machine. Similar to the DBF-SMM, a second Bloom filter can be employed to store a limited number of transition conditions δ such that its false positive probability is almost zero and its match results do not need verification. For this purpose, frequently encountered δ_C conditions can be stored in a second vector v_{FC}, where the associated next states $q_n = q_c + 1$ can be inferred easily.

Frequent δ_U conditions can not be stored in a Bloom filter vector because acquiring the associated next states via Bloom filter is a difficult task. In addition, the number of Δ_U transitions is expected to be low (please refer to Section 9.3); therefore a fraction of them can be exactly stored in a look-up table, namely Unconsecutive Transition Verification Unit (UTVU). The basic structure of the MBFbAC-wDFA is depicted in Fig.10.4 and the functionality of Monitor and Control Unit (MCU) is described in Algorithm 6. Note that each transition condition δ requires multiple memory accesses for the BF, UTVU and LUT queries, which are performed concurrently on hardware.

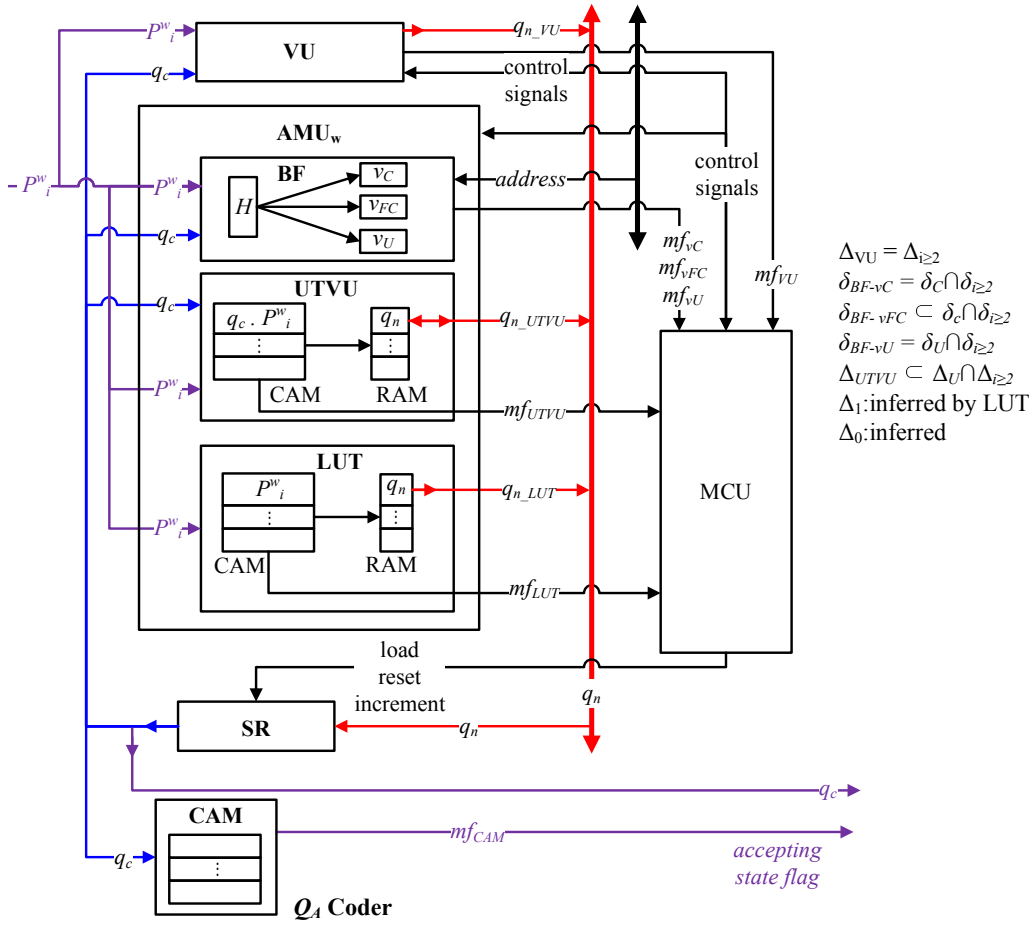


Figure 10.4: Basic structure of MBFbAC-wDFA.

MBFbAC-wDFA handles only constant length (w-byte) symbols. To perform string matching process, we need to handle 1 to w byte symbols. Therefore, w AMUs are employed, where each of which handles a predefined length of symbols as depicted in Fig. 10.5. If multiple matches of AMU occur, then the longest one must be considered

to determine the next state.

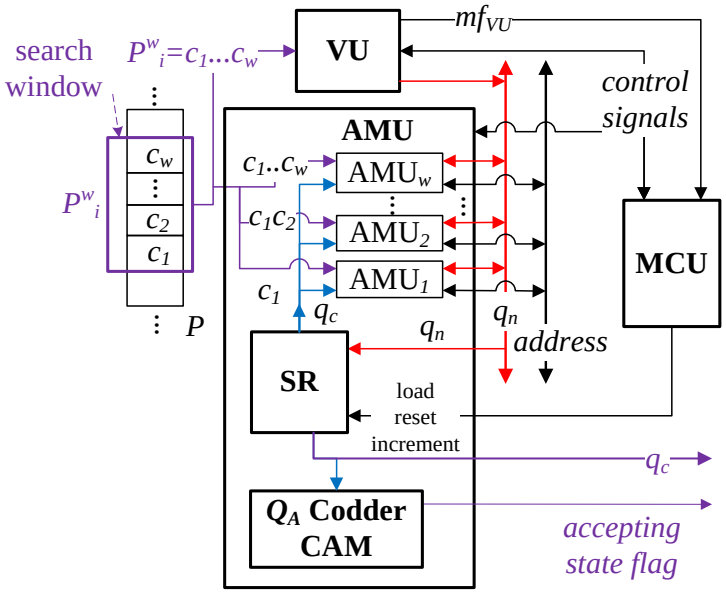


Figure 10.5: String Matching Module employing w AMUs.

To demonstrate the execution of the MBFb-wDFA, we use a SMM whose state transition graph and state transition rules are given in Fig. 9.13 and in Table 9.2, respectively. There are two automatons each of which scans one byte shifted part of the input string as shown in Fig. 10.7. Each automaton moves to the next state by considering the current state and consuming $w = 2$ bytes at once. For both of the automatons, the corresponding content of the BFs, VU, LUT and Q_A coder CAM are depicted in Fig. 10.6. The transitions to the states of depth-2 or more $\Delta_{i\geq 2} = \{(1, rs, 6), (2, e, 3), (4, s, 5)\}$ are stored in the VU. The consecutive transition conditions $\delta_C \cap \Delta_{i\geq 2} = \{(2, e), (4, s)\}$ and unconsecutive transition condition $\delta_C \cap \Delta_{i\geq 2} = \{(1, rs)\}$ are stored in the BFv_C and UTVU respectively. The input symbols $\{he, sh, hi\}$, which move the machines into the states of depth one (Q_1), and their associated next states are stored respectively into the CAM and RAM of the LUT, respectively.

The execution cycles of MBFbAC-wDFAs are demonstrated in Fig. 10.7 and 10.8, where the input string is $P = \{usherst \cdots\}$.

The zero next state of the transitions $(0, us, 0)$, and $(3, st, 1)$ are inferred by reset-

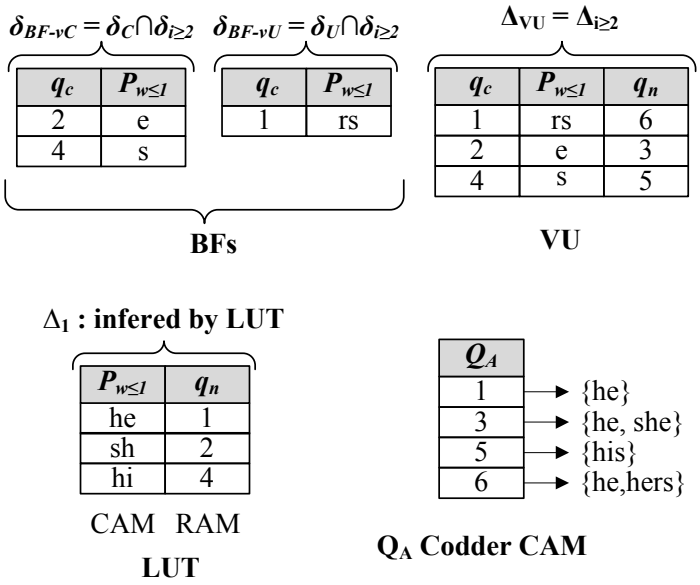


Figure 10.6: Content of each MBFbAC-DFA ($w = 2$) units each of which stores the string set $\mathcal{S} = \{he, she, his, hers\}$.

ting the state register to zero due to unmatch condition of the BFs, VU and LUT. Moreover, $(1, rs, 6)$ and $(2, er, 3)$ transitions are matched by the BFs and after the verification of each transition conditions, the associated next states are loaded from the VU. In addition, $(1, rs, 6)$ and $(2, er, 3)$ transitions are stored into the UTVU and BFv_{FC} respectively. The next states of the transitions $(0, he, 1)$ and $(0, sh, 2)$ (each of these transitions are $\in \Delta_1$) are loaded from the RAM of the LUT. Due to low false positive rate, BFs are not expected to match any false positive output under the case of unmatch condition of the LUT and UTVU.

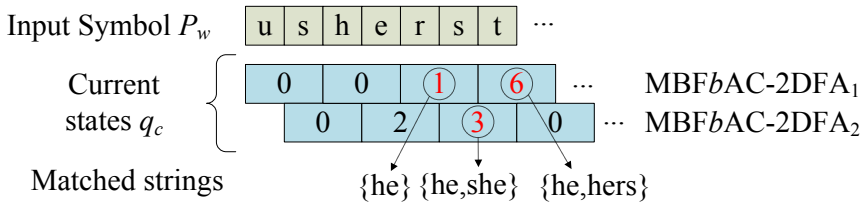


Figure 10.7: The execution of MBFb-2DFA based SMM matching the input string $P = \{usherst \cdots\}$ against the string set $\mathcal{S} = \{he, she, his, hers\}$. The red states depict the visited accept states.

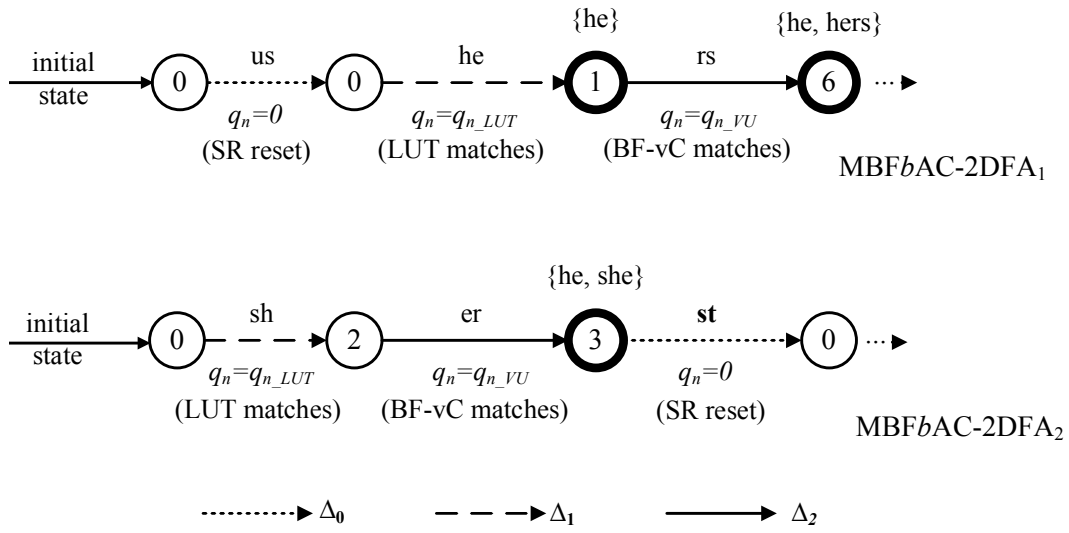


Figure 10.8: The execution of MBFb-2DFA based SMM matching the input string $P = \{usherst \cdots\}$ against the string set $\mathcal{S} = \{he, she, his, hers\}$. The bold vertices depict the visited accepting states.

10.2 Evaluations of the Proposed Architectures

The parameters of w-byte Aho-Corasick DFAs storing the signature string set of SNORT v2.9 are given in Section 9.3. In this part, we evaluate these parameters to the proposed machines and compare their normalized memory consumptions (NMC). The required amount of memory is calculated by the number of transitions stored in.

Table 10.1: Evaluation of the wDFA (improved) and BFbDFA machines storing the signature set of SNORT v2.9. The memory sizes are in bits and the number of shared hash functions is $k = 10$.

w	wDFA (improved)			BFwDFA	
	$\Delta_{LUT} = \Delta \setminus \Delta_0$, Δ_0 :inferred		NMC	$\delta_{BF} = \delta \setminus \delta_0$, Δ_0 :inferred	NMC
	$CAM_A + CAM_B$	$RAM_A + RAM_B$		BF	
1	$1.05 \cdot 10^9$	$1.32 \cdot 10^8$	$1,237$	$1.12 \cdot 10^8$	117
2	$1.15 \cdot 10^{10}$	$7.19 \cdot 10^8$	$12,758$	$6.48 \cdot 10^8$	677
3	$2.20 \cdot 10^{10}$	$9.18 \cdot 10^8$	$23,959$	$8.27 \cdot 10^8$	864
4	$4.96 \cdot 10^9$	$1.55 \cdot 10^8$	$5,346$	$1.49 \cdot 10^8$	156
5	$5.69 \cdot 10^9$	$1.42 \cdot 10^8$	$6,087$	$1.37 \cdot 10^8$	143
6	$6.30 \cdot 10^9$	$1.31 \cdot 10^8$	$6,711$	$1.26 \cdot 10^8$	132
7	$6.34 \cdot 10^9$	$1.13 \cdot 10^8$	$6,738$	$1.17 \cdot 10^8$	122
8	$7.68 \cdot 10^9$	$1.20 \cdot 10^8$	$8,146$	$1.24 \cdot 10^8$	129
9	$6.54 \cdot 10^9$	$9.09 \cdot 10^7$	$6,928$	$9.37 \cdot 10^7$	98
10	$6.82 \cdot 10^9$	$8.53 \cdot 10^7$	$7,214$	$8.79 \cdot 10^7$	92

Table 10.2: Evaluation of BFbAC-wDFA machines storing the signature set of SNORT v2.9. The number of shared hash functions is $k = 10$ and the memory sizes are in bits.

w	LUT		BF	Q_A codder	NMC
	$\Delta_{LUT} = \Delta_1$		$\delta_{BF} = \delta_{i\geq 2}$	CAM	
	CAM	RAM	v		
1	672	1,428	20,568,849	69,615	21.555
2	15,056	15,056	1,878,216	65,520	2.061
3	41,520	27,680	613,722	65,520	0.782
4	62,912	29,490	352,407	61,425	0.529
5	93,200	34,950	275,569	61,425	0.486
6	119,664	37,395	223,589	61,425	0.462
7	144,200	36,050	191,200	57,330	0.448
8	175,872	38,472	163,876	57,330	0.455
9	202,032	39,284	140,187	57,330	0.458
10	229,040	40,082	132,815	57,330	0.480

As depicted in Table 10.1, the improved naive implementations (wDFA-improved) have at least 10 times worse NMC values with respect to BFbDFA versions. Therefore, it is obvious that Bloom filters enable us to save significant amount of memory resources. Moreover, applying LUTs in BFbAC-wDFA to infer Δ_1 transitions deceases the number of transitions to be stored in memory and provides upto 330 times resource efficiency (Table 10.2). However, the execution speed (or input symbol consumption rate) of BFbAC-wDFA decreases significantly if BF matches frequently. Therefore, we have to add more memory sources to sustain the speed. That's why the SMM employing w MBFbAC-wDFAs necessitates slightly higher memory resources as given in Table 10.3. Note that, the memory efficiency of the machines employing CAMs to decode accepting states can be improved in such a way that the Q_A coder can be hardly coded into logic sources instead of a CAM.

10.2.1 Comparison of the Results with Other Related Studies

The performance results in terms of memory usage and complexity are summarized in Fig.10.9. In the literature, some of the studies are based on the different types of memories, such as RAM, CAM and TCAM. To achieve fair comparison, we also consider the hardware complexities of employed memories, therefore we add another parameter, namely, normalized memory complexity (NMCx). Each bit stored in SRAM,

Table 10.3: Evaluation of SMM constructed with w MBFbAC-wDFAs. SMM stores the signature set of SNORT v2.9. The number of shared hash functions is $k = 10$ and $v_{FC} = v_C * 50\%$. UTVU is limited to store 5 percentage of unsuccessive and depth-2 or more transitions, i.e., $|\Delta_{UTVU}| = |(\Delta_U \cap \Delta_{i\geq 2})| * 5\%$. The memory sizes are in bits.

w	LUT		UTVU		BF			Q_A codder	NMC
	$\Delta_{LUT} = \Delta_1$		$\lvert\Delta_{UTVU}\rvert$		$\delta_{BF} = \delta_{i\geq 2}$			CAM	(CAM/Logic)
	CAM	RAM	CAM	RAM	v_C	v_U	v_{FC}		
1	–	$4,352$	$9 \cdot 10^6$	$1.1 \cdot 10^6$	$1.3 \cdot 10^6$	$19.3 \cdot 10^6$	$1.3 \cdot 10^6$	$69,615$	$33.58/33.50$
2	$14,968$	$15,056$	$1.1 \cdot 10^6$	$68,553$	$641,956$	$1.24 \cdot 10^6$	$641,956$	$65,520$	$3.95/3.88$
3	$41,088$	$27,680$	$242,227$	$10,093$	$431,712$	$182,010$	$431,712$	$65,520$	$1.50/1.43$
4	$61,648$	$29,490$	$57,336$	$1,792$	$317,941$	$34,466$	$317,941$	$61,425$	$0.92/0.86$
5	$87,768$	$34,950$	$36,150$	904	$258,185$	$17,384$	$258,185$	$61,425$	$0.79/0.72$
6	$109,120$	$37,395$	$32,652$	680	$210,504$	$13,085$	$210,504$	$61,425$	$0.71/0.64$
7	$127,736$	$36,050$	$30,145$	538	$180,106$	$11,094$	$180,106$	$57,330$	$0.65/0.59$
8	$151,488$	$38,472$	$24,506$	383	$155,984$	$7,892$	$155,984$	$57,330$	$0.62/0.56$
9	$168,496$	$39,284$	$22,831$	317	$133,651$	$6,535$	$133,651$	$57,330$	$0.59/0.53$
10	$185,176$	$40,082$	$21,840$	273	$127,188$	$5,627$	$127,188$	$57,330$	$0.59/0.53$

CAM and TCAM requires 6, 10 and 20 transistors, respectively [28, 31]. Therefore we apply 10/6 and 20/6 times penalties for the usage of each CAM and TCAM bit. On the other hand, only the memory consumptions per SMM are considered for fair comparison.

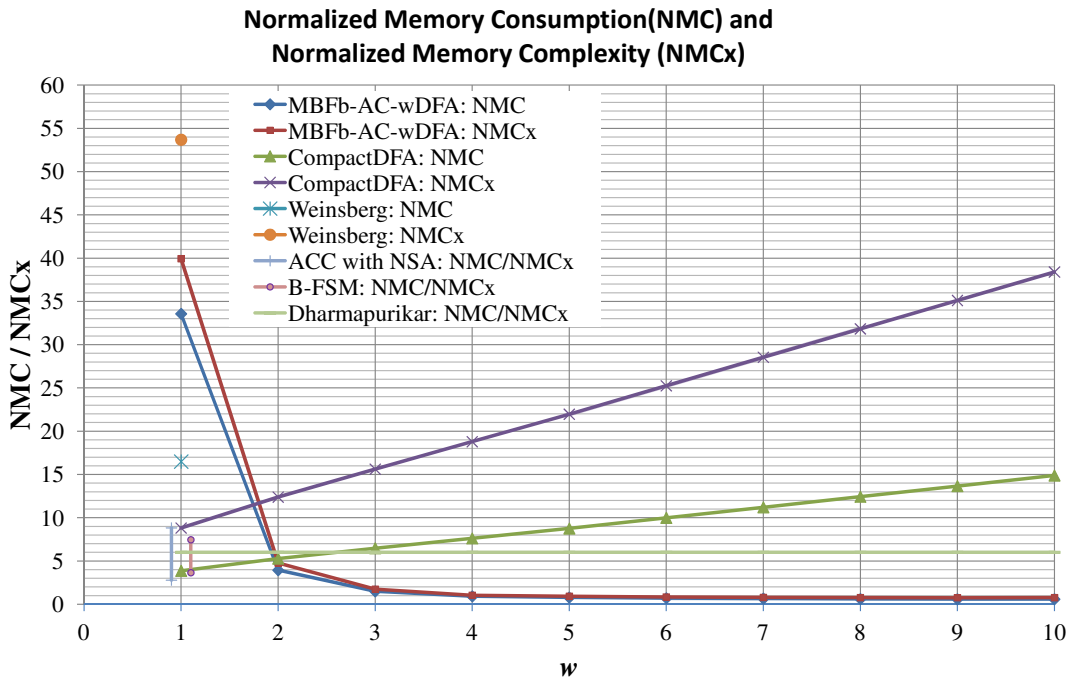


Figure 10.9: Summary of comparison results.

Although Barr et al. [12] modified the string set to include only strings of length 5 bytes or more, we assume that they apply all 6423 strings having 30 bytes length in average. Moreover, for the unevaluated values $w \geq 6$, we estimate the parameters by assuming linear increase as a function of w. Furthermore, only the implementations considering minimization of the number of entries and having variable width entries, where each rule is stored with exactly the number of bits required to encode it, are inspected. The results depict that the overall memory sizes of CompactDFA [12] is in line with w because of increase in code width per state and the number of entries. Hence, NMC and NMCx increases linearly with w.

The proposed machine by Weinsberg et al. [41] is based on the on an earlier work of [17] and provides significant improvement over this machine in terms of memory. However, their studies cover only automatons consuming single bytes, i.e. $w = 1$.

Both of the studies [36] and [39] are based-on only RAMs; hence their NMC and NMCx values are same. For ACC with NSA proposed in [36], NMC values are calculated for 1785 strings from SNORT with total 29.0KBytes. For 2-way and 2 subsets configuration 256KByte RAM is needed, whereas 81KByte RAM is required for 8-way with 16 subsets. Besides, while storing 1.5K strings with 25.2KByte size, 188KBytes and 92KBytes of memories are needed for 4 and 16 groups, respectively in B-FSM [39].

16 hash functions are considered for the study proposed by Dharmapurikar et.al. in order to store 2259 strings having 6.94bits in average in [16].

Although the worst NMC and/or NMCx values belong to our proposed machine MBFb-AC-wDFA for $w = 1$, we are able to achieve the best performance when $w \geq 3$. Therefore, we can apply high number of parallel SMMs to achieve higher scan rates.

10.3 Related Work

An automaton should determine which state to move according to the current transition condition. Naive implementation procedure exactly stores all possibilities, namely, one transition rule per each transition condition. This results in unimplementable large amount of on-chip memory requirement. For instance, to naively store the signature strings of SNORT v2.9 by AC-wDFA, 401MByte CAM and 54MByte RAM are needed (for $w = 1$). In such implementations, some of the transitions, for example $\Delta \setminus \Delta_0$ transitions, can be stored explicitly and the others, for example Δ_0 transitions, can be inferred. While traversing through the automaton, if current transition conditions is not found in LUT, then the Δ_0 transition logic is triggered to return the automaton to the initial state q_0. Even in this automaton, 132MByte CAM and 17MByte RAM are needed (for $w = 1$).

In general, AC-like DFAs are time-efficient but space-inefficient and NFA versions are time-inefficient but space-efficient. This is because of NFAs may need multiple transitions (multiple cycles) to consume one input symbol, since the matching operation needs to explore multiple paths in the automaton to determine whether the input

matches any string in the set. Although they may require large amount of matching time, they have relatively small number of transitions. However, DFAs have deterministic execution speed (one cycle needed to consume one input symbol) but they generate large number of state transition rules. Therefore, there has been an intensive effort in order to compactly implement AC-like DFAs that can fit into small but fast on-chip memories [38, 34, 21, 36, 33, 9, 39, 12]. A common optimization that significantly reduces the DFA memory size is the removal of Δ_0 and/or Δ_1 transitions [38, 34, 21, 36, 33], which is based on the observation that these transitions constitute a large fraction of overall transitions Δ and removing them provides significant memory reduction.

In [21], Kumar et.al. introduce a new representation for DFAs, called the Delayed Input DFA (D^2FA), which substantially reduces space requirements. The idea of D^2FA is based on the observation that many states in a DFA often share some set of transitions. Therefore; several transitions of the automaton can be replaced with a single *default transition*, which reduce the amount of memory requirements. The replacement operation is based on a technique used in AC-automaton [8]. The approach dramatically reduces the number of transitions between states by more than 95% for a collection of regular expressions drawn from current commercial and academic systems. Default transitions are triggered when there is no corresponding transition for the input symbol. Although an input symbol leads to a single transition between states in a classic DFA, a D^2FA may require multiple default transitions to consume a single character, which slows down the automaton. In order to compensate this side effect, the automaton is implemented with multiple on-chip memories and processors, each of which is uniformly occupied and accessed, which results in high throughput.

In [9], similar to D^2FA, by exploiting redundancy in transitions, a DFA is represented in a compact way. In fact, this approach is based on the observation that automatons recognizing regular expressions have large amount of transitions lead back to the low depth states. The proposed automaton is called FastFA. An alphabet reduction scheme is also employed to achieve higher compaction rates. However, the level of compression achieved is similar to the original D^2FA [21], while providing a better worst-case speed. Because, regardless of the maximum length of the default transitions, the proposed automaton needs up to $2n$ transitions when processing an

input string of length n. The authors achieved compression rates higher than 99% for all sets tested.

Van Lunteren observed that a large fraction of state transitions are Δ_0 and Δ_1 transitions [39, 36]. Similar to our approach, he successfully reduced Δ_0 and Δ_1 transitions having at most 256 transitions per state (for $w = 1$) [39] . The reduction is achieved by using the priority approach; Δ_0 and Δ_1 transitions are assigned to low priorities of 0 and 1 respectively. Therefore; these transitions can be combined to single transition rules. BaRT-based Finite State Machine technology (B-FSM)[39] provides compact hash-based data structure and one state transition per operation cycle. After that, they extends automatons to support essential features of regular expression and reduces common state transitions [40].

Eliminating some of the other transitions ($\Delta_{i\geq 2}$), namely *cross transitions*, are considered in [36]. To do this, they try to partition the string set to many unrelated smaller sets (similar to [39]), handled by multiple small AC-DFAs, so as to avoid common sub-strings and their transition rules. Besides, they propose to use next states to accessing the transition rules in memory. The proposed machine consumes 81KByte (9.5MByte) RAM memory to store 1.8K (50K) rules (total 29K(4.44M) characters) NMC is 2.79 (2.13) for SNORT (ClamAV) set. However, the optimization results in a significant disadvantage of traversing multiple states. It is observed that in the worst-case, the processing of the input strings needs a traversal of up to 31 pointers, which causes 22X performance degradation with respect to the the average-case [33]. In order to improve the worst-case performance, Bloom filters are employed to jump the automaton to next state by bypassing the failure transitions, which yields over 3X performance improvement in [33].

The state coding can be performed as desired as long as all states are assigned to unique codes. Barr et. al. [12] employed this observation to achieve a compressed DFA, namely CompactDFA. In fact, the proposed automaton is a kind of generalization of [39] and [36] that eliminates all cross transitions. They encode the states in such a way that all transitions to a specific state are represented by a single prefix storing a set of current states. The overhead of this process is the higher number of bits per state. For Snort, only 17-bits are needed per state for arbitrarily distributed

state codes, while 36-bits are required in [12]. Therefore they limit the number of current states stored in prefix. Besides, in order to achieve high scan rates, they also need to divide the transition rules into multiple TCAMs.

In [24] Le et. al. propose an algorithm to preprocess the string set in such a way that all the nonleaf strings are emerged with their leaf strings. In order to improve the NMC, the strings are divided into segments, where each of them are executed with cascaded Binary Search Trees. By this approach, for fixed-length strings of SNORT, the algorithm decreases the NMC from 1,45 to 1,07. A pipelined search tree is also presented in order to one look-up operation per clock cycle to increase the scan rate. By employing these two techniques, 1,32 NMC and 3,2Gbps scan rate are achieved for SNORT set (the results belongs to a dual-pipeline architecture with dividing the strings into 24-byte long segments).

Algorithm 6 Control Unit(CU) function of MBFbAC-wDFA.

Initially: The machine is in state q_0 (reset value of SR)
Condition: Main *if* functions are concurrent
Condition: If multiple units of UTVU, v_C, v_U, v_{FC} or LUT are employed to handle symbols which are shorter than w bytes, assign higher priority to the longest one.
For each queried input symbol ${P^w}_i$:
if UTVU or v_{FC} matches **then** ▷ *Exact* matching occurs
    if UTVU matches **then** ▷ UTVU has higher priority than v_{FC}
        $q_n = q_{n\_UTVU}$ ▷ Load the next state from UTVU
    else $q_n = q_c + 1$ ▷ v_{FC} matches, increment the current state
    end if
else ▷ v_C and v_U have lower priority than UTVU and v_{FC}
    if v_C or v_U matches **then** ▷ *Approximate* matching occurs
        wait VU ▷ Due to verify the BF matches
        if VU matches **then** ▷ True positive output for BFs
            $q_n = q_{n\_VU}$ ▷ Load the next state from VU
            if v_C matches **then**
                store transition condition (q_c, ${P^w}_i$) into v_{FC}
            else ▷ v_U matches
                store transition (q_c, ${P^w}_i$, q_n) into UTVU
            end if
        end if
    else if LUT matches **then**
        $q_n = q_{n\_LUT}$ ▷ Δ_1 transition occurs
    else $q_n = q_0$ ▷ Return to initial state
    end if
end if

CHAPTER 11

CONCLUSIONS AND FUTURE WORK

In this thesis, we present Double Bloom Filter String Matching Module (DBF-SMM) as a novel Bloom-Filter based SMM architecture. The first Bloom Filter of DBF-SMM stores the entire string set that is searched and its no-match results are correct because of the zero false negatives of the Bloom Filters. The significant feature of DBF-SMM is the second Bloom Filter that queries the input strings concurrently with the first one. The second filter stores a small string set and consequently it has almost zero false positive probability. This small set is dynamically updated to contain the strings that are likely to appear in the current input. Hence, the second filter detects most of the matching strings without further verification. As one possible method, we suggest updating the content of the second Bloom Filter with the most recent matches.

Our analysis results show that the response time of the DBF-SMM is decreased down to the response time of the Bloom Filter component provided that the content of the second Bloom Filter is always covering the matches in the current input string. Furthermore, our FPGA implementation results show that even parallel implementations of DBF-SMM are feasible to implement on contemporary FPGAs potentially achieving up to 10 Gbps scan rates. Besides, the DBF-SMM is modeled in SystemC environment and the required functionality of the proposed architecture is verified by applying 10 different types of input traffic.

We also employed the idea behind the DBF-SMM for implementing fast Deterministic Finite Automaton with efficient use of hardware resources. The resulting architecture and its implementation provide a viable solution for applications such as variable size string matching, regular expression matching and speech recognition.

Researchers have been working to achieve multi-gigabit scan rates, frequently by trying to increase the automaton speed and decrease the amount of memory needed. To do this, we use the features of the classic AC machine behind the string matching process and employ these features to fast hardware blocks. Automaton-evaluated string matching is generally memory-intensive process, which limits performance in commodity hardware resources. In the thesis, we provide background information of AC string matching algorithm and propose BF-based machines which reduce the amount of memory required significantly and improve its performance on real hardware implementations. The signature string set of SNORT v2.9 is also implemented as AC-based wDFAs and the parameters of the generated automatons are analyzed. Based on this analysis, we carefully group the state transition rules according to their next state characteristics and map them to appropriate hardware components. Instead of storing all of the rules, only a small portion are exactly stored in memory and the other transitions are inferred. As a result, the proposed architecture enables us to represent entire data structure of AC machine with small amount of memory. Additionally, high scan rates are achieved thanks to parallelism of hardware blocks, where groups of transition rules are queried concurrently.

BF based approaches provide high memory efficiency and high execution speed; however, the main disadvantage of this approximate matching techniques is that the scan rate decreases with high positive matches. To cope with this problem, we employ the idea behind the DBF-SMM, where the slow verification process is eliminated for frequent state transition rules. As a result, the proposed SMM (based on MBFb-AC-wDFA) consumes only 0.53 bytes of memory per byte for signature string set of SNORT v2.9 and sustains 10 Gbps scan rate on average under the case of high positive rate.

The proposed hardware-based approaches all exploit a high degree of parallelism by representing transitions of automaton by the parallel logic resources available in FPGA or ASIC devices. However, it might not be acceptable in systems where the string sets needed to be updated frequently, which necessitates quickly re-synthesize and update the logic behind SMM functionality. Therefore, SMMs which rely on memory rather than logic are often more desirable as they provide higher degree of flexibility and programmability. On one hand, the logic behind our proposed ma-

chines is not needed to be re-synthesized frequently. In fact, re-synthesis procedure is not needed until a pre-defined size limit of the string set stored in the machine. On the other hand, only the associated memory contents can be modified with pre-calculated values to support changes in the string set.

The proposed architectures may be evaluated further with other string sets extracted from other sources such as ClamAV [3], BRO [2], Roget [4], etc. Moreover, the variable size string matching architecture can be mapped into commercially available FPGAs and off-chip memories in order to depict the feasibility and the scan rate. Also, instead of assigning CAM devices for each substring length, a single TCAM can be used. At this point, the penalty in terms of Normalized Memory Consumption (NMC) and Normalized Memory Complexity (NMCx) should be calculated. Furthermore, similar to DBF-SMM, MBFb-AC-wDFA can be modeled in SystemC and its functionality can also be verified with different types of input traffics.

The performance of the proposed architectures can be improved further; the scan rate, for example, can be increased in the normal traffic conditions by employing heuristic mechanisms. To do this, appropriate methods published in the literature should be inspected carefully and modified properly to be used in the proposed architectures.

In addition to this, higher memory efficiency can be achieved with sharing multi-port memories for parallel SMMs having the same memory contents.

REFERENCES

[1] Accellera systems initiative. http://www.accellera.org. Accessed: 2014-08-12.

[2] Bro: Network monitoring and network intrusion detection system. http://bro-ids.org/. Accessed: 2014-08-12.

[3] Clam antivirus: an open source (gpl) anti-virus toolkit for unix. http://www.clamav.net. Accessed: 2014-08-12.

[4] Roget: System hacking general password crackers wordlists. http://packetstormsecurity.org/files/31989/roget-dictionary.gz.html. Accessed: 2014-09-28.

[5] Snort: A free lightweight network intrusion detection system for unix and windows. http://www.snort.org. Accessed: 2014-08-12.

[6] Virtex-7 fpgas data sheet: Dc and switching characteristics, April 2014.

[7] Xilinx:a digital programmable logic device (pld) company, April 2014.

[8] M. J. C. Alfred V. Aho. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[9] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In R. Yavatkar, D. Grunwald, and K. K. Ramakrishnan, editors, *ANCS*, pages 145–154. ACM, 2007.

[10] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[11] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10), October 1977.

[12] A. Bremler-Barr, D. Hay, and Y. Koral. Compactdfa: Scalable pattern matching using longest prefix match solutions. *Networking, IEEE/ACM Transactions on*, 22(2):415–428, April 2014.

[13] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[14] H. Chen, Y. Chen, and D. Summerville. A survey on the application of fpgas for network infrastructure security. *Communications Surveys Tutorials, IEEE*, 13(4):541 –561, quarter 2011.

[15] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom filters. *Micro, IEEE*, 24(1):52 – 61, jan.-feb. 2004.

[16] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *Selected Areas in Communications, IEEE Journal on*, 24(10):1781 –1792, oct. 2006.

[17] Y. Fang, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *ICNP*, pages 174–183. IEEE Computer Society, 2004.

[18] S. Geravand and M. Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 57(18):4047–4064, 2013.

[19] H. Kim, H.-S. Kim, and S. Kang. A memory-efficient bit-split parallel string matching using pattern dividing for intrusion detection systems. *Parallel and Distributed Systems, IEEE Transactions on*, 22(11):1904 –1911, nov. 2011.

[20] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, pages 155–164, 2007.

[21] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 339–350, New York, NY, USA, 2006. ACM.

[22] A. Lakshmikantha, C. Beck, and R. Srikant. Impact of file arrivals and departures on buffer sizing in core routers. *Networking, IEEE/ACM Transactions on*, 19(2):347 –358, april 2011.

[23] R. P. Laufer, P. B. Velloso, and O. C. M. B. Duarte. A generalized bloom filter to secure distributed network applications. *Comput. Netw.*, 55(8):1804–1819, June 2011.

[24] H. Le and V. Prasanna. A memory-efficient and modular approach for large-scale string pattern matching. *Computers, IEEE Transactions on*, 62(5):844–857, May 2013.

[25] P.-C. Lin, Y.-D. Lin, Y.-C. Lai, Y.-J. Zheng, and T.-H. Lee. Realizing a sub-linear time string-matching algorithm with a hardware accelerator using bloom

filters. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(8):1008 –1020, aug. 2009.

[26] Y. Meng, W. Li, and L.-F. Kwok. Towards adaptive character frequency-based exclusive signature matching scheme and its applications in distributed intrusion detection. *Computer Networks*, 57(17):3630 – 3640, 2013.

[27] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33:31–88, March 2001.

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

[29] A. Papadogiannakis, D. Antoniades, M. Polychronakis, and E. Markatos. Improving the performance of passive network monitoring applications using locality buffering. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2007. MASCOTS '07. 15th International Symposium on*, pages 151 –157, oct. 2007.

[30] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos. Tolerating overload attacks against packet capturing systems. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[31] A. Patwary, B. Geuskens, and S. Lu. Low-power ternary content addressable memory (tcam) array for network applications. In *Communications, Circuits and Systems, 2009. ICCCAS 2009. International Conference on*, pages 322–325, July 2009.

[32] M. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *Computers, IEEE Transactions on*, 46(12):1378 –1381, dec 1997.

[33] G. S. Shenoy, J. Tubella, and A. González. A performance and area efficient architecture for intrusion detection systems. In *IPDPS*, pages 301–310. IEEE, 2011.

[34] G. S. Shenoy, J. Tubella, and A. González. Hardware/software mechanisms for protecting an ids against algorithmic complexity attacks. In *IPDPS Workshops*, pages 1190–1196. IEEE Computer Society, 2012.

[35] I. C. Society. Ieee standard for standard systemc language reference manual. *IEEE Std. 1666-2011*, January 2012.

[36] T. Song, W. Zhang, D. Wang, and Y. Xue. A memory efficient multiple pattern matching architecture for network security. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages –, April 2008.

[37] K.-K. Tseng, Y.-C. Lai, Y.-D. Lin, and T.-H. Lee. A fast scalable automaton-matching accelerator for embedded content processors. *ACM Trans. Embed. Comput. Syst.*, 8:19:1–19:30, April 2009.

[38] 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.

[39] J. Van Lunteren. High-performance pattern-matching for intrusion detection. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–13, April 2006.

[40] J. Van Lunteren and A. Guanella. Hardware-accelerated regular expression matching at multiple tens of gb/s. In *INFOCOM, 2012 Proceedings IEEE*, pages 1737–1745, March 2012.

[41] Y. Weinsberg, S. Tzur-David, D. Dolev, and T. Anker. High performance string matching algorithm for a network intrusion prevention system (nips). In *High Performance Switching and Routing, 2006 Workshop on*, pages 7 pp.–, 2006.

[42] N. Weng, L. Vespa, and B. Soewito. Deep packet pre-filtering and finite state encoding for adaptive intrusion detection system. *Computer Networks*, 55(8):1648 – 1661, 2011.

[43] L. Yang, R. Karim, V. Ganapathy, and R. Smith. Fast, memory-efficient regular expression matching with nfa-obdds. *Computer Networks*, 55(15):3376 – 3393, 2011.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Zengin, Salih
Nationality: Turkish (TC)
Date and Place of Birth: 25.04.1979, Ankara
Marital Status: Married
Phone: +90 506 282 47 64
Phone: +90 312 590 91 85
Fax: +90 312 210 13 92

EDUCATION

Degree	Institution	Year of Graduation
M.S.	Electrical and Electronics Engineering Dept., METU	2006
B.S.	Electrical and Electronics Engineering Dept., METU	2003

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
2004-	TÜBİTAK SAGE	Chief Expert Researcher