行政院國家科學委員會專題研究計畫　成果報告

Web Switch 系統的設計與製作(2/2)

計畫類別：個別型計畫
計畫編號：NSC91-2219-E-009-032-
執行期間：91 年 08 月 01 日至 92 年 07 月 31 日
執行單位：國立交通大學電信工程學系

計畫主持人：李程輝

計畫參與人員：曾德功、郭英哲、謝坤宏、羅天佑、吳銘智

報告類型：完整報告

處理方式：本計畫可公開查詢

中　華　民　國 92 年 10 月 1 日

行政院國家科學委員會補助專題研究計畫成果報告

※※※※※※※※※※※※※※※※※※※※※※※※※※
※
※ Web Switch 系統的設計與製作（2/2） ※
※
※
※※※※※※※※※※※※※※※※※※※※※※※※※

計畫類別：☑個別型計畫 □整合型計畫

計畫編號：NSC 90－2213－E－009－089－

執行期間：91 年 08 月 01 日至 92 年 07 月 31 日

計畫主持人：李 程 輝 國立交通大學電信工程學系 教授

本成果報告包括以下應繳交之附件：

□赴國外出差或研習心得報告一份

□赴大陸地區出差或研習心得報告一份

□出席國際學術會議心得報告及發表之論文各一份

□國際合作研究計畫國外研究報告書一份

執行單位：國立交通大學電信工程研究所

中　華　民　國　92 年 7 月 30 日

目錄

LIST OF TABLE

Table
page

LIST OF FIGURE

Figure
page

Abstract

The purpose of this project is to investigate and implement some key functions of a web switch. After a thorough survey, we determined to study and implement the “persistence” and the Quality of Service (QoS) features. The reason is that many features such as delayed binding, load sharing, and TCP splicing had been studied extensively and already implemented in existing products. The persistence feature means that multiple connections of a session are connected to the same server so that a dialogue between a client and server can be continued. This is considered an important feature for e-commerce because a user may issue multiple transactions during his/her visit of a commercial web site. The states of the visiting client are stored in the server it was connected to and thus the session can be continued smoothly if succeeding connections of the client are connected to the same server. The persistence feature can also improve system performance because authentication and encryption are necessary in e-commerce and new keys have to be generated if the client is connected to a different server. The QoS feature means that some sessions are guaranteed a delay bound to receive service. This is also an important feature for e-commerce because customers can be satisfied with different QoS requests.

To accomplish the persistence and QoS features, mechanisms that establish connections based on IP address and/or cookie are necessary. In other words, fast IP address lookup and string matching are important techniques. In the following, we describe our research results on IP address lookup and string matching separately.

Keywords: Broadband Internet, Quality of service, IP address lookup, string matching, e-commerce, cookie.

中文摘要

本年度計劃的目標在於探討並完成 web 交換機系統所需的關鍵技術。經過一番透徹的研讀相關研究報告與文獻，我們決定深入研究並完成網路連線持續性與服務品質保證之特性。由於其他重要的特性，諸如延遲時間限定、負載平衡、TCP 封包接合等，均已經被大量研究且已經實現在一些問世的產品上，所以本計劃著重在網路連線持續性與服務品質保證的關鍵技術研究與實現。所謂持續性功能，乃在於同一個 session 支援多條連線且連接至相同的伺服器，使得客戶端與伺服器端的對話(dialogue)可以連續。這項特性對電子商務是一項非常重要的功能，主因在於一個使用者連接到一個電子商務網站可能同時發起多個交易行為。而此到訪客戶的狀態會儲存在伺服器內，若此客戶後續的連線也是連接到此伺服器上，則 session 將可以很順暢的連通。此持續性特質亦可改善系統的效能，主因在於認證與加解密所需要的金鑰可以分享同一個 session 的金鑰。在電子商務上，認證與加解密是必須的，而當客戶端連接到不同的伺服器時，就需要先產生金鑰，而產生金鑰的程序是相當耗時的，會降低系統效能。服務品質是本計劃將著眼的另一項特性，主要在於保證 session 所收到服務的延遲上限。由於不同的使用者滿足於不同的服務品質需求，所以服務品質亦是電子商務上另一項重要的特質。

為了完成持續性與服務品質特性，我們必須要透徹研究基於 IP 位址或 cookie 所建立連線的機制。換句話說，就是快速的 IP 位址查詢與 string matching 機制。於此計劃中，我們將個別描述 IP 位址查詢與 string matching 機制的研究結果。

關鍵詞：寬頻網際網路、服務品質、IP 位址查詢、電子商務

Part I:

Fast IP Routing Table Lookup

I.1. ABSTRACT

IP routing table lookup has been considered as a major bottleneck for high-speed web switch routers. In the past few years, several data structures and related algorithms have been developed to accomplish high-speed routing table lookup. In particular, an efficient algorithm, called binary search on prefix lengths, was designed by grouping prefixes of identical lengths into individual tables and applying hashing technique in these tables to find matching prefixes. The time complexity of binary search on prefix lengths is $\lceil \log(W+1) \rceil$ assuming W-bit address. In Part I of this project, we propose a multi-way search algorithm on prefix lengths to improve the average lookup performance of the binary search scheme without sacrificing its worst-case search performance. The proposed scheme is so simple that it basically does not increase the complexity in constructing the search tree and in memory requirement. Through experiments on real backbone routing tables, we found that the improvement can be more than 37% for most tables and 21% for one table.

I.2. INTRODUCTION

Because of the explosive growth of Internet traffic, the performance of IP routing table lookup is becoming critical for high-speed routers to provide satisfactory services. As such, many researches developed in the past few years new algorithms to accomplish high-speed routing table lookup [1]-[10]. Some of the algorithms compress the routing table with sophisticated data structures so that a processor can perform routing table lookup in its cache [1]-[5] and some others use simple data structures with special hardware to assist the search process [6]-[7]. In general, sophisticated data structure renders difficulty in table update and simple data structure may require a large amount of memory.

Since there are two parameters N (the number of route prefixes) and W (the number of address bits) in the IP routing table lookup problem, the time complexity of a search algorithm may depend on either N or W. For example, linear search is a simple algorithm of time complexity N. Since insertion and deletion of a route prefix is very simple in linear search, it is widely used when N is small. To reduce the search time, one can encode a route prefix as a range [4]. Assuming that W is 6, then a route prefix like 11* is actually a range addresses from 110000 to 111111. When route prefixes are encoded as ranges, one can apply binary search for table lookup. As was shown in [4], the search time complexity is $\log(2N)$ for N route prefixes. Although the search time is largely reduced with binary search, it is more difficult to insert or delete a route prefix.

Search algorithms with time complexity depending on W are likely to be better choices when the number of route prefixes N is large. Radix trie [5] is a well-known example of routing table lookup algorithm whose time complexity depends on W. To reduce the search time, an interesting algorithm, called binary search on prefix lengths, proposed in [8] can search for the longest matching prefix for an incoming packet with time complexity $\lceil \log(W+1) \rceil$. In this search algorithm, the route prefixes are organized into tables according to their lengths and uses a hashing technique to look for the matching prefix in each of these tables. In fact, the binary search on prefix lengths is deemed an efficient solution for routing table lookup. However, each hash probe takes at least one memory access, which is significant at gigabit speeds. Thus, several variants have been developed to improve the average lookup performance by employing a weighting function to optimize the binary search tree pattern [9]. Unfortunately, they are not applicable to routing table lookup because either the worst-case lookup performance is sacrificed or the tree pattern

could be altered frequently as route prefixes change. There are several other interesting routing table lookup algorithms. One can find a good survey of these algorithms in [10].

In Part I of this report, we present a multi-way search on prefix lengths to improve the average lookup performance of standard binary search on prefix lengths without scarifying the worst-case search time and storage requirement. This algorithm is simple and basically does not increase the complexity in constructing the binary tree. We show through experiments with real routing tables that the improvement could be more than 37% for most tables and 21% for one table.

In Part I, the rest of this report is organized as follows. In Section I.2, we briefly review the concept of binary search on prefix lengths proposed in [8]. An example is provided to explain the operation of such an algorithm. Our proposed algorithm, named multi-way search on prefix lengths, is presented in Section I.3. In Section I.4, we compare the lookup performance of these two schemes with some collected routing tables. Finally, we draw conclusion in Section I.5.

I.3. BINARY SEARCH ON PREFIX LENGTHS

The IP routing table lookup problem can be described as follows. Given an incoming packet's destination IP address, find the longest matching prefix among a set of route prefixes. Since every route prefix length is variable, it is difficult to determine how many bits of the destination address should be taken into account when compared with the route prefixes. To overcome this problem, an interesting approach has been proposed that allows a sequential search on length dimension. Such an approach organizes the route prefixes in different tables according to their

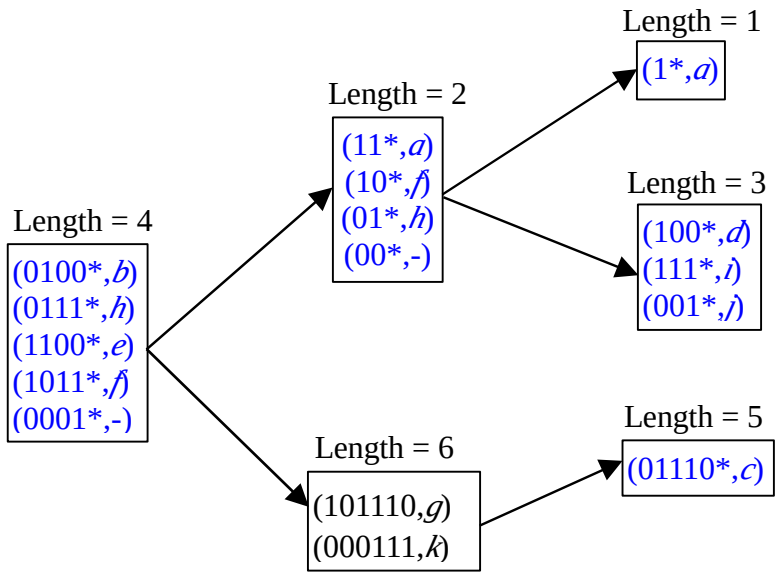
lengths and uses a hashing technique to look for the matching prefix in each of these tables. The search strategy is to probe these tables in sequence starting from the one holding the longest prefixes and terminate whenever a match is found. Assuming that the address length is W and a perfect hash function is applied, this approach requires up to W probes.

In order to reduce the search time, a binary search on prefix lengths was proposed in [8]. In this algorithm, markers and pre-computation are two important mechanisms. Markers are used to guide the search process while pre-computation avoids back tracking. More concretely, individual tables form a binary tree based on route prefix lengths to enable binary search. To guide the binary search process, a route prefix in table i places a marker in an ancestor node if and only if the node representing table i lies in the left sub-tree of the ancestor node. (Table i denotes a table that consists of route prefixes with length i.) To avoid back tracking, one has to pre-compute the longest matching prefix for each marker. The longest matching prefix of a marker is a matching route prefix that is shorter than the marker but longer than any other matching route prefixes. Note that it is possible that a route prefix R and a marker M both match the query address. Based on the above definition, the longest matching prefix is R. The search process starts to probe the root node. If a match is found, we move to its left child. Otherwise, probe its right child. The search process ends after a leaf node is probed. Clearly, the number of hash probes is $\lceil \log(W+1) \rceil$. Since a route prefix can generate up to $\log W$ markers, the worst-case memory requirement is $N \log W$ with a set of N route prefixes.

In Fig. I.1, we use an example route prefixes to illustrate this scheme. The content of individual hash tables in binary tree form is also shown in Fig. I.1. In this example, we assume that there are 12 route prefixes and W is equal to 6. In this

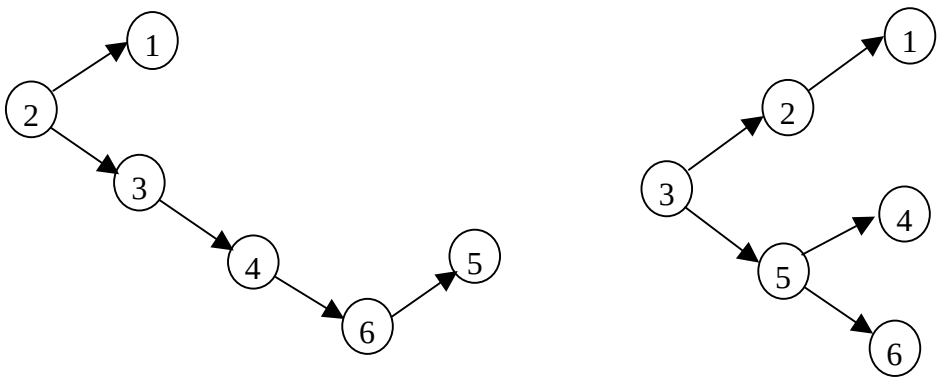
figure, an entry (*X, Y*) represents a route prefix or a marker *X* and its corresponding longest matching prefix *Y*. In case the longest matching prefix is null, *Y* is replaced with "-". In Fig. I.1, one can find that route prefix *c* (= 01110*) places one marker 0111* in hash table 4 and the pre-computation for this marker is route prefix *i*. Further, another marker 0001* in hash table 4 is generated by route prefix *l* (= 000111) but the corresponding pre-computation result is null because there is no route prefix that is a prefix of 0001*. Assume that we are looking for the longest matching prefix of the address 110011. The search process starts with table 4, moves to tables 6 and 5 in sequence, and then terminates. The longest matching prefix found is *f*.

Route prefixes	
a	1*
b	0100*
c	01110*
d	100*
e	1100*
f	10*
g	101110
h	01*
i	111*
j	001*
k	000111



In fact, binary search on prefix lengths is deemed an efficient solution for routing table lookup. However, each hash probe takes at least one memory access, which is significant at gigabit speeds. Thus, several variants of binary search on prefix lengths have been developed to reduce the average number of hash probes. For example, a heuristic approach presented in [9] changes the tree-shaped search pattern according to the address space covered by all route prefixes in a hash table. That is, a table with more addresses covered by route prefixes contained in the table is placed

closer to the tree root. The binary tree of this approach for the example route prefixes is depicted in Fig. I.2(a). In this figure, node *i* represents a table that consists of all prefixes with length *i*. Clearly, this introduces asymmetry into the binary tree. While reducing the average lookup performance, some query packets could degenerate towards linear search, which is undesirable. Also, incremental update (insertion or deletion of a route prefix) may require reconstructing the search tree in order to optimize the average performance. Another heuristic approach presented in [9] is to build a useful asymmetric tree based on a weighting function which is defined to be the number of entries in a table. As a consequence, a table with more route prefixes is placed closer to the tree root. However, the worst case bound has to be satisfied. Consider the example route prefixes illustrated in Fig. I.1. Assume that *W* is equal to 6 and thus the worst case bound is 3. The weights of tables 1, 2, 3, 4, 5, and 6 are 1, 2, 3, 2, 1, and 2, respectively. Therefore, table 3 is chosen to be the tree root. It is not hard to see that the worst case bound is violated if we choose either table 4 or table 6 to be the left child of table 3. Therefore, table 5 is the only choice to be the left child of table 3. The result is shown in Fig. I.2(b). Clearly, such an approach also needs to reconstruct the tree after some route prefixes are inserted or removed. In the next section, we present a simple algorithm, called multi-way search on prefix lengths, to improve the average performance without sacrificing the worst case performance and changing the tree-shaped pattern as route prefixes are altered.



(a) Maximize addresses covered. (b) Maximize entries while keeping the worst case bound.

Fig. I.2. Asymmetric search trees generated respectively by two heuristic approaches for the example route prefixes.

I.4. MULTI-WAY SEARCH ON PREFIX LENGTHS

Recall that, for the binary search on prefix lengths, a route prefix in table i will place a marker in an ancestor node if and only if the node representing table i lies in the left sub-tree of the ancestor node. As an example, consider a 21-bit long route prefix F in the IPv4 protocol. Suppose that a complete binary tree is constructed based on prefix lengths as shown in Fig. I.3(a). It is clear that route prefix F in node 21 places markers in node 20 and 16. Assume that no other route prefixes create the same markers as F. For an address A whose longest matching prefix is F, the search process in binary search on prefix lengths will traverse nodes 16, 24, 20, 22, and 21 in sequence. That is, five hash probes are needed. However, we found that the search time can be reduced if there are links in between nodes 16 and 20 and between nodes 20 and 21 so that the search process can directly move through these three nodes to find the longest matching prefix. To achieve this, we maintain a multi-way search tree structure.

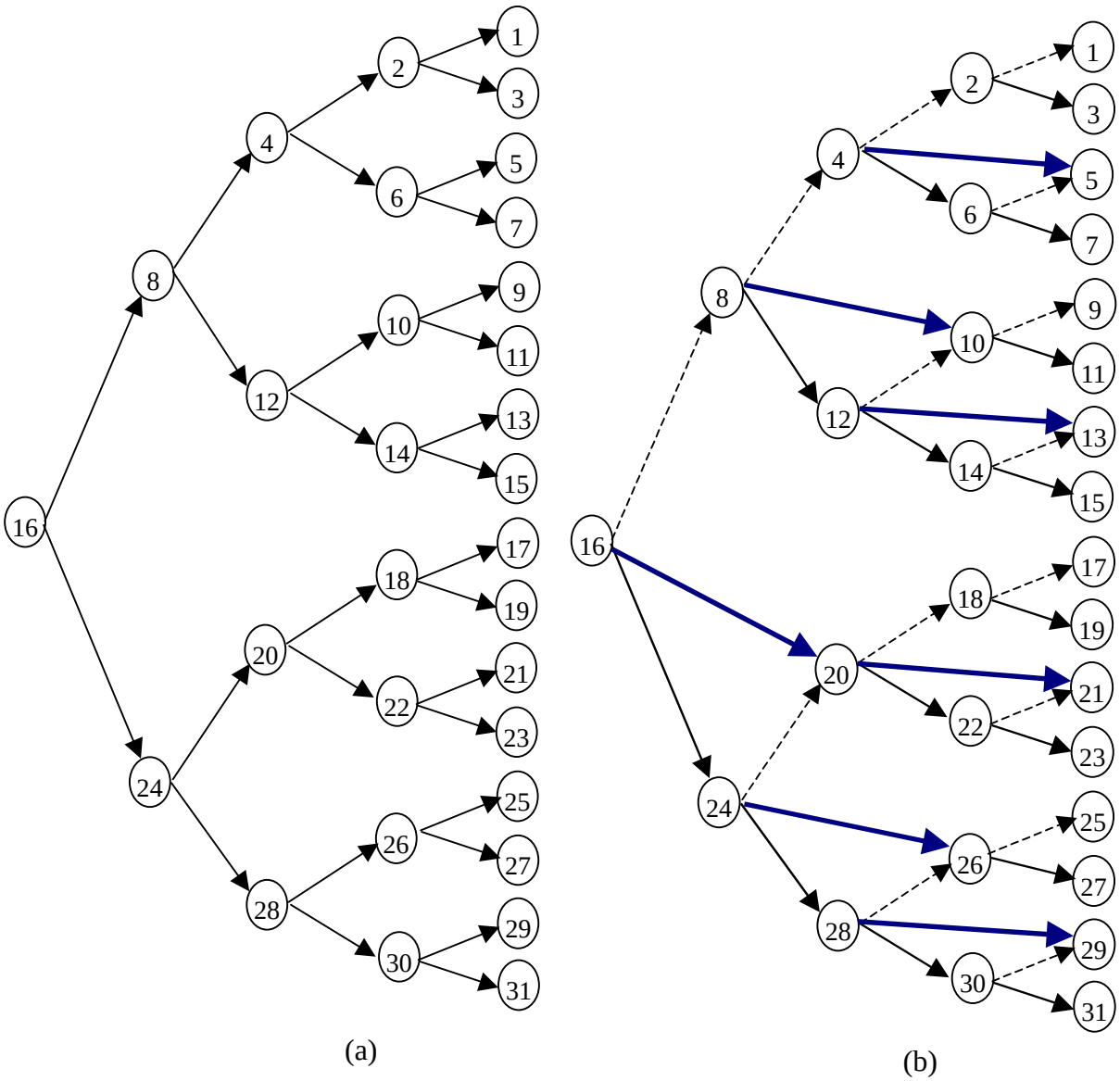


Fig. I.3. (a) Binary search and (b) 3-way search on prefix lengths in IPv4 protocol.

For ease of description, we consider the 3-way search tree structure shown in Fig. I.3(b) in the following. Similar to binary search on prefix lengths, we first construct binary tree and then recursively create markers for route prefixes. Two types of markers, i.e. *L*-marker and *R*-marker, are employed in the 3-way tree structure. Thus, each marker consists of a prefix value and a 1-bit state to indicate which type this marker is. To build the third branch of an internal node and create markers for each route prefix, we follow the conditions described below. Consider a node i and its left sub-tree with root node j. Let G_l^j denote the set consisting of node j and all the nodes in its left sub-tree. Similarly, let G_r^j represent the set of nodes in the right

sub-tree of node *j*. For every route prefix in a node that belongs to G_l^j, we place an *L*-marker for this route prefix in node *i*, as depicted in Fig. I.4(a). For any route prefix in a node belonging to G_r^j, we place an *R*-marker for this route prefix in node *i* if there does not exist an *L*-marker having identical prefix value in that node. In addition, we build a link from node *i* to the root node of the right sub-tree of node *j*, i.e. the third branch of node *i*, as shown in Fig. I.4(b). For example, refer to Fig. I.3(b), a route prefix in node 26 will create an *L*-marker in node 16 and an *R*-marker in node 24 if no other *L*-markers in that node have the same prefix value. A link is constructed from node 24 to node 26 so that, when an *R*-marker is found in node 24, the search process can directly move to probe node 26. Notice that *L*-marker is identical to that generated by binary search on prefix lengths and *R*-maker is used to speedup the search time.

The search process starts to probe the root node of the entire 3-way search tree. If a "no match" is returned, we move to its rightmost child. If an *R*-marker is matched, we move to its middle child. Otherwise, probe its leftmost child. The process terminates whenever a leaf node is probed. The pseudo program of the 3-way search algorithm is described in Fig. I.5.

To simplify incremental update, we suggest constructing a complete tree that consists of all possible prefix lengths (like the one shown in Fig. I.3(b)). To insert a new route prefix, one can simply place the route prefix in the appropriate table, generate necessary markers and perform pre-computations for these markers. If the inserted route prefix generates an *L*-marker that is identical to an *R*-marker generated by other prefixes, then the original *R*-marker becomes void. Notice that some existing markers may have to change their best matching prefixes when a new route prefix is inserted. In other words, we have to verify whether or not the inserted route

prefix is the best matching prefix of a marker if the original best matching prefix of

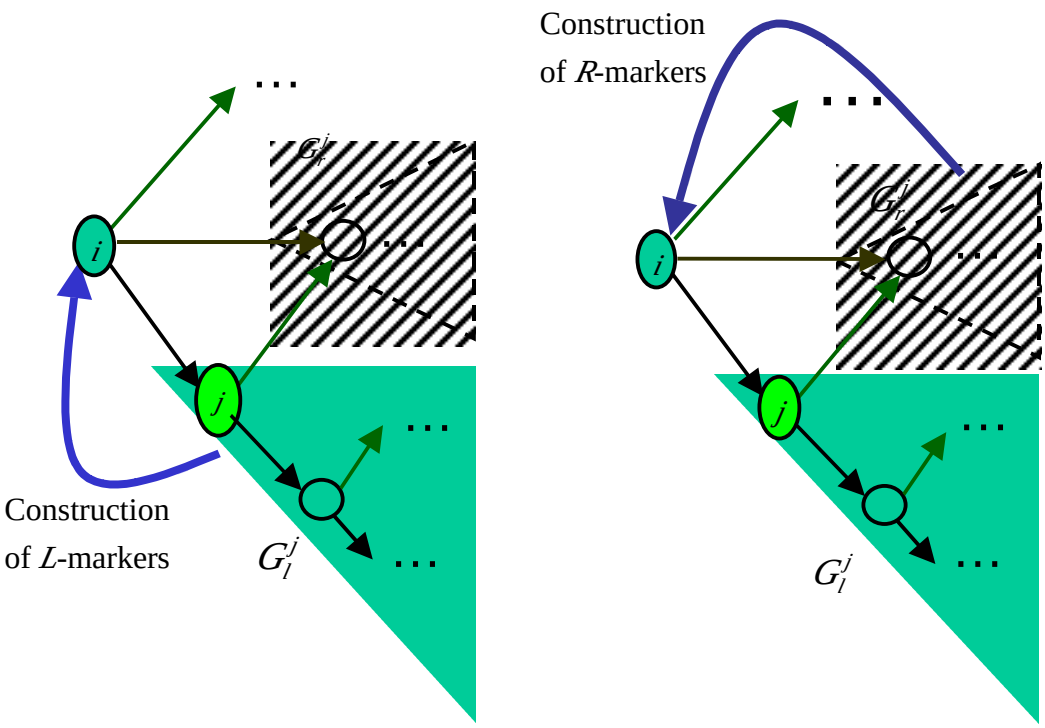


Fig. I.4. (a) Construction of L-marker in node i for every route prefix in a node belonging to set G_l^j. (b) Construction of R-marker in node i for any route prefix in a node belonging to set G_r^j.

the maker is a prefix of the inserted one. One can utilize the original tree to accomplish this. For example, suppose a route prefix F of length L is to be inserted. A necessary condition for a marker to change its best matching prefix is that F is a prefix of the marker. Assume that the length of the marker is $L+k$. The marker can be found as long as all 2^k possible combinations are searched. Another method for updating the best matching prefixes of markers is to first find all route prefixes that have F as their prefix and then change best matching prefixes for corresponding markers.

To delete a route prefix, one should remove the route prefix, its markers, and update the best matching prefixes of those markers that have the deleted route prefix

as their best matching prefixes. Assume that a route prefix F of length L is to be deleted. Also, let H be the longest route prefix that is a prefix of F. Any marker that has F as the best matching prefix must change it to H. All the markers that potentially have to change their best matching prefixes can be found with the same process as F is to be inserted. To determine H, one can search all tables of length smaller than L.

Given a 3-way search tree and suppose that each node in the tree is a hash table consisting of route prefixes of identical length.

Input: Query address ***A***

Output: Longest matching prefix ***F*** for query address ***A***

Variable: Tree's node ***X***

Initialization: Set node ***X*** as the root node of the 3-way search tree and ***F*** as the default route prefix or null.

While (node ***X*** is not null) **{**

Probe node ***X*** to find the matching route prefix for query address ***A***;

If ("no match" in node ***X***) **{**

node ***X*** ← rightmost child of node ***X***;

}

else {

If (an *R*-marker is matched in node ***X***) **{**

Update the longest matching prefix ***F*** as the pre-computation result of the *R*-marker;

node ***X*** ← middle child of node ***X***;

}

else {

Update the longest matching prefix ***F*** as the pre-computation result of the *L*-marker;

node ***X*** ← leftmost child of node ***X***;

}

}

}

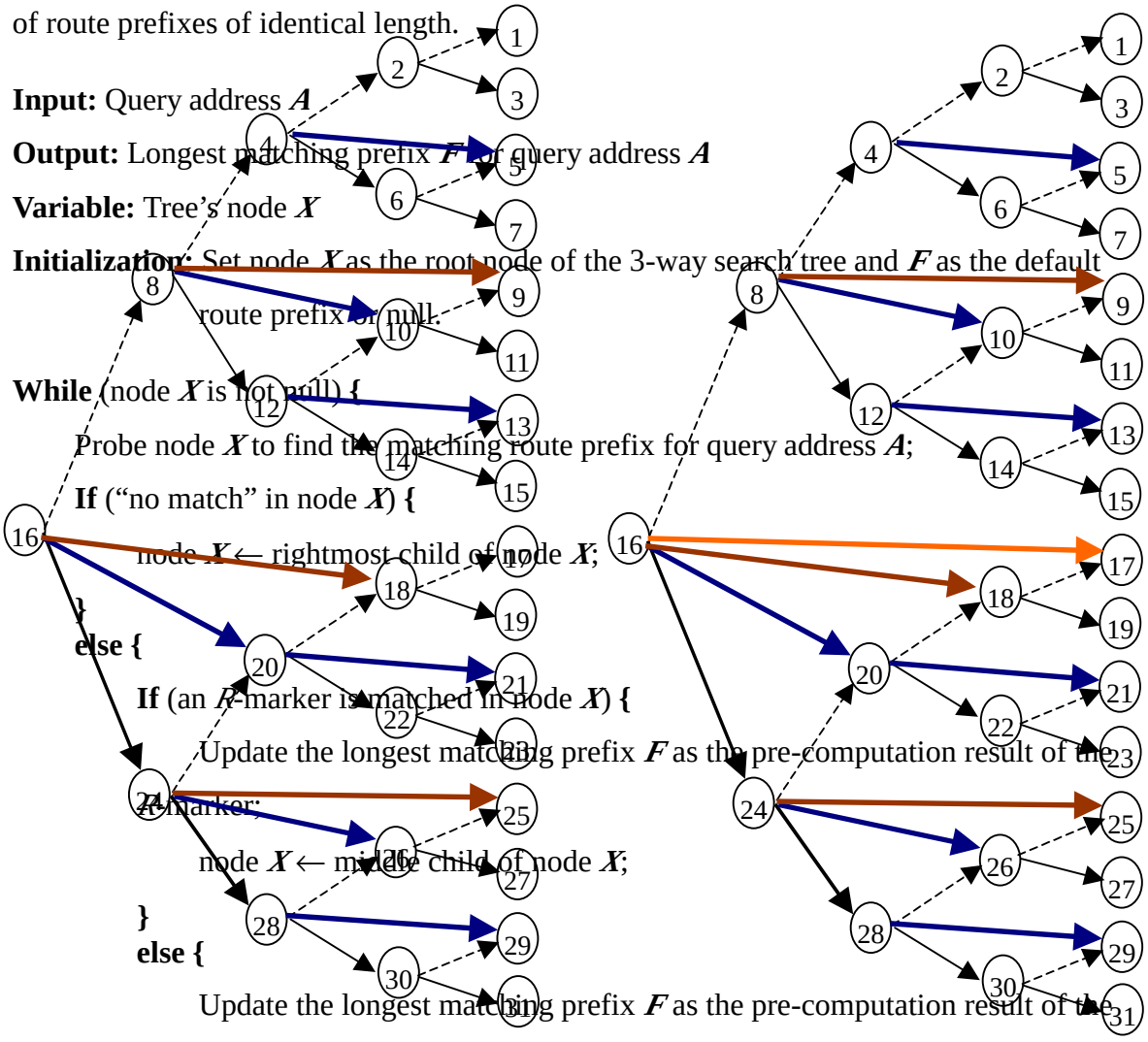


Fig. I.6. (a) 4-way and (b) 5-way search on prefix lengths in IPv4 protocol.

Fig. I.5. The pseudo program of the 3-way search algorithm.

Obviously, the concept of 3-way search tree can be generalized to an arbitrary *k*-way tree structure. Fig. I.6 shows the 4-way and 5-way search trees for IPv4 route prefixes. A directed link from node 16 to node 18 means that a route prefix or a marker of length 18 places a marker in node 16 and there is no other route prefixes or markers of length longer than 19 that places the same marker in node 16. Of course, for the 4-way (and 5-way) search tree, we have to use two bits to distinguish three

(and four, respectively) different markers and guide the search process. It is not hard to see that the maximal k is five for 32-bit IPv4 addresses and seven for 128-bit IPv6 addresses. Basically, our proposed algorithm is so simple that it does not increase the complexity in constructing the search tree and in memory requirement. In the next section, we compare the average lookup performance of our proposed scheme with that in binary search on prefix lengths through experiments on real backbone routing tables.

I.5. EXPERIMENTAL RESULTS

In this section, we experimentally evaluate the performance of our proposed algorithms and compare with the binary search scheme on prefix lengths in terms of the average number of hash probes. Four backbone routing tables in PAIX, AADS, MAE, and MCVAX are collected from the IPMA project for the experiments [13]-[14], as shown in Table I.1. To measure the average number of hash probes required, we consider querying all possible addresses that are created respectively from the non-overlapping intervals delimited by given route prefixes. Note that each route prefix has two end points and the collection of all the end points will divide the entire address space into a set of non-overlapping intervals. The lookup results for these algorithms are summarized in Table I.1. The results show that the average number of probes required in the binary search on prefix lengths is close to the theoretical bound in the worst case, i.e. $\log W$. However, our proposed 3-way search algorithm significantly reduces the average lookup time by a factor of more than 21% for MCVAX routing table and 37% for other tables. As can be seen from the numbers, the further improvement of either 4-way or 5-way search is not as

significant as the improvement obtained from the 3-way search. Therefore, our suggestion is to use the 3-way search algorithm to reduce the update complexity when route prefixes change.

It should be noted that the improvement in the MCVAX routing table is less than those in other tables. This is because that there is nearly half of 32-bit route prefixes and thus nearly half of querying addresses needs the worst-case number of probes to find the longest matching prefix.

Table I.1. Comparison of average number of probes required in the binary search and our proposed multi-way search algorithms on prefix lengths for four backbone routing tables.

Routing table	Number of prefixes	Average number of probes			
		Binary search	3-way search	4-way search	5-way search
PAIX	21936	4.8611	3.0569	2.8377	2.7911
MAE	42290	4.7053	2.9759	2.8466	2.8016
AADS	40723	4.7284	2.9846	2.8253	2.7943

I.6. CONCLUSION

We have presented in this report a simple multi-way search algorithm on prefix lengths to improve the average performance of the binary search scheme while keeping the same worst-case performance in IP routing table lookup. Through experiments on real backbone routing tables, we found that the improvement can be more than 21% for MCVAX routing table and 37% for other tables. Since the gain increases as the depth of the search tree increases (which is induced by longer address length), we believe that our proposed algorithm achieves more improvement for 128-bit IPv6 routing table lookup. Further, one can apply our proposed algorithm to improve the average search time for tuple space-based multi-field packet

classification [11]-[12].

Part II:

String Matching Algorithms and its Hardware Implementation

II.1. ABSTRACT

String matching is a performance bottleneck in the pattern-based load balancing or QoS guarantee in a web switch. This problem can be solved by a suitable hardware of string matching. Thus, we survey the related algorithms and hardware implementation in Part II of this project, and try to propose the suitable algorithm and architecture for the string matching. According to our survey and analysis, we conclude that the multiple pattern and wildcard are the mandatory requirements for string matching, and bit parallel are the most suitable algorithm for hardware implementation.

II.2. INTRODUCTION

Pattern-based load balancing or QoS guarantee requires the deep content inspection of packet and the content is variable in the length and start position, thus the string matching is the suitable technique for this content inspection. From the previous related papers [15-16] and based on our observation, we can classify the string matching into the following taxonomy:

Dynamic programming algorithms: use the dynamic programming algorithm to compute the pattern and text distance of similarity. This kind of algorithms is no need the preprocessing phrase for the patterns.

Filter algorithms: use the heuristic skipping technique to save the comparison of characters. This kind of algorithms needs the preprocessing phrase to build the skipping tables.

Automaton algorithms: Automaton algorithms transfer the patterns into the automaton in preprocessing phrase, and in the search phrase it traverse the automaton to find a match. The automaton traversing is implemented as a lookup processing from the transition table with the corresponding state and character.

Bit parallel algorithms: This kind of algorithms uses bit vector to simulate the automaton, it converts the characters of patterns to the bit occurrence table in the preprocessing phrase. In the searching phrase, it lookups current characters of text in occurrence table and perform the bitwise operation with a state mask. If there is a matching the corresponding bit of state mask will be zero or one.

The string matching is the major performance bottleneck, thus the hardware solution is useful to boost the speed. In this project, we also survey existing the hardware implementations of string matching algorithms. According to our study, there are three architecture have been proposed. They are (1) systolic architecture, (2) pipeline and parallel architecture, (3) reconfigurable architecture. However, the existing hardware implementation is not suitable for string matching algorithms at all. In addition, we have the analysis of string matching requirements, the related algorithms and its hardware implementation so that we can conclude that the bit parallel string-matching algorithm is a suitable algorithm and hardware implementation for string matching.

Part II of this project is organized as follows. Section II.2 studies some important string matching algorithms as our taxonomy. Section II.3 surveys existing

hardware implementation of string matching algorithm. Section II.4 discusses the requirements of string matching, and proposed the suitable hardware architecture. Finally, Section II.5 is our conclusion.

II.3. STRING MATCHING ALGORITHM

There are many string matching algorithms have been widely discussed in a recent several decade, but only some on-line string matching algorithms that preprocessing of pattern are suitable for load balancing and QoS guarantee, the preprocessing of text is suitable for information retrieval. In this section we study related string matching algorithms in detail and classify them as our previous taxonomy.

A.1 DYNAMIC PROGRAMMING ALGORITHM

Dynamic programming algorithm was proposed in the [17], which uses the dynamic programming array to compute the result of string matching. At first, it forms an n*m matrix of text and pattern, n is the length of text and m is length of pattern. This algorithm intends to find the distance-similarity between pattern and text, if the distance is less, the similarity is more. The common distance value is the edit distance; the edit distance is the mismatching number between the pattern and of text. One edit distance value can be reduced by one operation of insertion, deletion or substitution. The following is a formula of edit distance computation, D denote edit distance matrix, and i, j are the index of text x and pattern y. If x_i is equal y_j in character, then current $D(i,j)$ is set to the left upper $D(i-1,j-1)$ value, otherwise $D(i,j)$ is set to the minimum value of the left upper $(i-1,j-1)$, left $D(i-1,j)$ or upper $D(i,j-1)$ value that plus one.

$$D_{i,0}=i$$

$$D_{0,j}=j$$

$$D(_{i,j}) = \text{if } (x_i=y_j) \text{ then } D_{i-1,j-1} \text{ else } 1 + \min(D_{i-1,j},D_{i,j-1},D_{i-1,j-1})$$

Dynamic programming algorithm computes edit distance from left top to right bottom cells, and fill the cell one by one. The final result is the most right bottom cell. This algorithm has worst case O(n) and not need the preprocessing of pattern. Because its matrix is fixed size, and its best case is its worst-case time complexity. This algorithm is only suitable for the application with short pattern and text.

Fig. II.1 is a dynamic programming example; the text is “search” and pattern is “seerh”. The most right bottom cell is 2, which means the pattern and the text having two differences of edit distance. That needs two operation of insertion, deletion or substitution to let the pattern to be equal the text.

		s	e	a	r	c	h
	0	1	2	3	4	5	6
s	1	0	1	2	3	4	5
e	2	1	0	1	2	3	4
e	3	2	1	1	2	3	4
r	4	3	2	2	1	2	3
h	5	4	3	3	2	2	2

Fig. II.1. A Dynamic programming example.

A.2 AUTOMATON ALGORITHMS

Automaton algorithms search the text by traversing the automaton graph. The automaton is built from the pattern. For the simple automaton algorithm, it builds the minimal deterministic automaton to recognizing the single pattern; the pattern is converted into an automaton in this preprocessing phrase. Moreover, for the character that is not in the string path, automaton will has a state transition to the initial state, but if there is a repeated prefix, it will has a state transition by skipping the prefix state. Fig II.2 illustrates an automaton for pattern “ababc”. The searching phrase is traversing its automata if there is existed a path. When the final state is reach, then a string matching is report. It has worst case O(n) for the searching

time, and it has the same states as the length of pattern. Thus, this preprocessing time is O(m) in preprocessing time.

In addition to the basic automaton algorithm, we also survey three advanced automaton algorithms in this section, that includes Aho-Corasick[18], reverse factor[19] and multiBDM[20] algorithms.

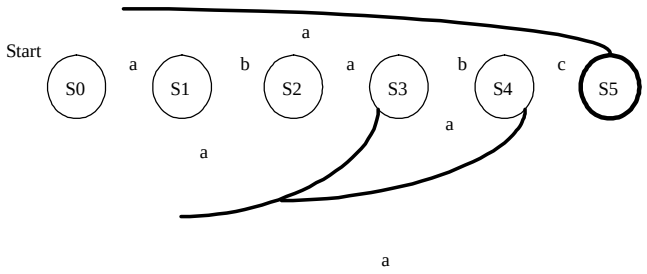


Fig. II.2. Simple automaton for pattern “ababc”.

A.2.1. AHO-CROASICK ALGORITHM

Aho-Croasick is a multiple patterns string matching algorithm. In the preprocessing phrase, it builds three tables from the patterns, that includes goto table: g(state, t_i), failure table: f(state) and output table: output(state). The “goto” table is easy to build, each character of pattern is an edge of automaton and the patterns share the same edge if they have same prefix. The output table is built from marked the state that it should output a string matching. When a mismatch is occurred, the next state will be the state by looking the corresponding failure table. Failure table is computed from that the length of current longest suffix that is also a prefix. This means it can skip the prefix comparison if a prefix is found.

In the searching phrase, each text character is inputted to g(state, ai), if its next state output(state) is not empty, it means that the pattern is matching. If its next state

fails to be found, g(f(state), ai), and it checks output(state) again. This algorithm has worst case O(n) for searching multiple patterns. The following is an example, the patterns are {he, she, his, hers}, if the input text is “ushers”. In the goto table g(4, e)=5, The output(5) has “she” and “he”. But when g(5, r), there is not next state in goto table, then a failure is occurred. Sine failure table f(5)=2, it will lookup the goto table again, but g(2, r)＝8 , and output(8) is not output, thus no string matching is reported in this case.

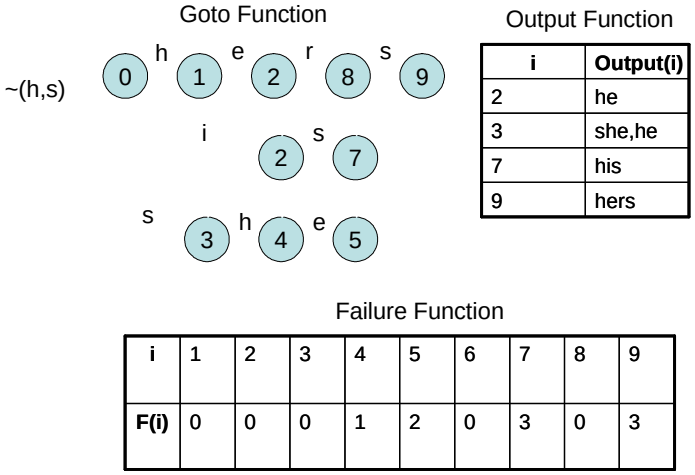


Fig. II.3. An Aho-Corasick example.

A.2.2. Reverse Factor Algorithm

Reverse factor algorithm is also called suffix automaton algorithm; the suffix automaton is a DAWG (Directed Acyclic Word Graph) that match a string from right to left in a text window, and the text windows is slide from left to right along with the input text. The suffix automaton is built from reversed pattern in the preprocessing phase. Fig. II.4 demonstrates an suffix automaton for pattern P ={aabbaab}, firstly we reverse pattern P^r ={baabbaa}, and each character is a edge, then we add an initial state and empty a transitions to all other states. This is a NFA and showed on the top automaton of Fig. II.4. The second step, we build the bottom automaton of Fig. II.4 using subset construction.

In the searching phase, the suffix automaton is used to save the comparison when a mismatch occurs in the text window. This algorithm read the text characters of the text window from right to left, and moves along the automaton. And the path in the automaton from the initial state to the final state is a knowing the longest prefix, that used to guarantee safe skip for a mismatch. When there is a longest prefix in a mismatch path, this algorithm aligns with the longest prefix, not skips all text window. For example, if the text is the "abbabaabbaab", the above suffix automaton has a mismatch path "baaba", there is a longest prefix aab in current path, thus the text window is slide to align the aab prefix.

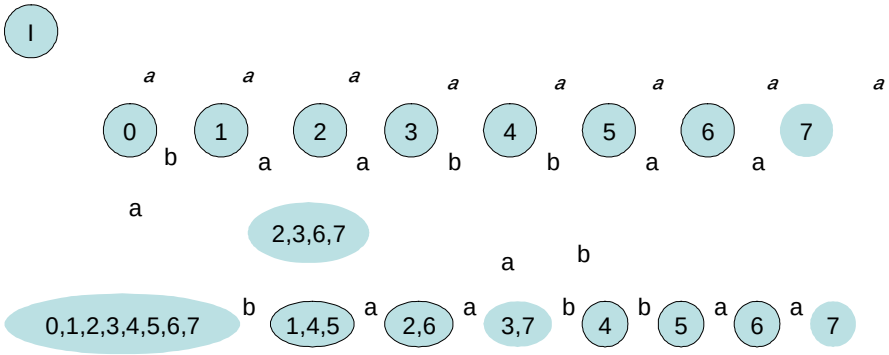


Fig. II.4. A suffix automaton example.

A.2.3. MULTIBDM ALGORITHM

The multiple backward DAWG (MultiBDM) algorithm combines the suffix automaton and Aho-Corasick automaton, thus it have advantage to use their heuristic techniques. In the preprocessing phrase, the Aho-Corasick automaton and suffix automaton will be built as the fundamental algorithms.

In search phrase, the MultiBDM algorithm scans DAWG from right to left to skip some comparison in a text window, then scans Aho-Corasick automaton to checks the pattern match. There are two cases in the scanning DAWG. The case 1: Scan_DAWG is stopped before critical position *critpos*, if Scan_DAWG found a

longest prefix *pre*, then it updates *critpos* and repeat the Scan_DAWG again. The case 2: Scan_DAWG reaches the critical position *critpos*, then it does Scan_AC from critical position *critpos*, and read until end window *pos*. If the recognized *pre>lim/2*, then Scan_AC continue to read, otherwise restart a Scan_DAWG. The notation *lmin* is the minimal length of patterns, *pos* is end of the window, *pre* is the longest prefix that reads by Scan_DAW, *u* is a prefix of the window that reads by Scan_AC and *critpos* is critical position, end of *u*.

Fig. II.5 is an example for patterns {abbb, ababbba}, the right automaton is the Aho-Corasick automaton and the left automaton is the suffix automaton. The suffix automaton is built from the reverse cut patterns {bbba, baba}, and the dash line is the failure link in the Aho-Corasick automaton. The following three steps show a searching process over the text “abbabbababbba”.

Step 1: [abba]bbababbba, *pos* = 4, *critpos* = 0. Initially, it does Scan_DAWG to read {a,b}, if we fail to read {b},and *pre* = {a}, and *critpos* is not reached. The *pre* < *lmin/2*, thus restart a Scan_DAWG.

Step 2: abb[abba]babbba, *pos* =7, *critpos* = 4. Firstly, it does Scan_DAWG to read {a, b}, if we fail to read {b}, and new *pre* = {a}, *critpos* is not reached. The *pre* < *lmin/2*, thus restart a new Scan_DAWG.

Step 3: abbabb[abab]bba with *pos*=10, *critpos* = 7. Firstly the Scan_DAWG reads [b, a, b, a], *pre* = {a, b, a, b}, and *critpos* is reached. It belongs to case 2, and Scan_AC reads {a, b, a, b}, and recognized *pre >= lmin /2*, thus continue to read {b, b} and report a match, and then read {a} and report second match again.

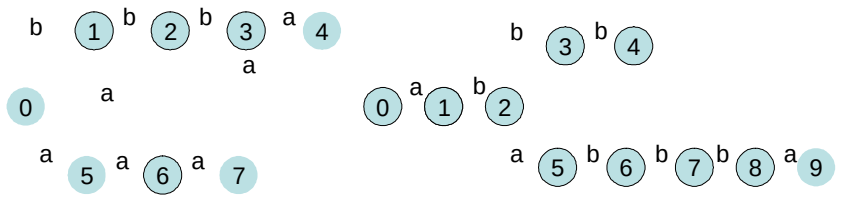


Fig. II.5. A MultiBDM example.

A.3 REGULAR EXPRESSION ALGORITHM

Regular expression (RE) is a generalized string description with basic string, kleene star (*), concatenation and union (|). Each RE has a corresponding finite automaton (FA), the FA can be nondeterministic or deterministic, the nondeterministic finite automaton (NFA) allows more then one next transition in the state transition, and its conversion from RE to NFA requires m states. The deterministic finite automaton (DFA) allows only one next transition, DFA may have up to 2^m states, and thus the DFA requires larger space to store its transition state than NFA.

If we use the DFA for string matching, in preprocessing phrase, a DFA is built from the pattern by using the subset construction, and because this method is famous in formal language textbook, we omit the detail introduction in here. In the searching phrase, we read the text and traverse the DFA to find a match, if a final state is reached, it we output a match. If we use the NFA, it needs to extra technique to distinguish the multiple transitions for an input. There are two NFA constructions have been proposed, Thompson's construction [21] and Glushkov's construction, Thompson's construction produces up to $2m$ states and it is not ε free ε NFA. Glushkov's construction produces exactly m+1 states and it is ε NFA. Thus the Glushkov's construction is superior to the Thompson's construction. And we demonstrate the Glushkov's construction only in section in Fig. II.6.

Glushkov's construction has m+1 states, and marks the states with order number initially, then it uses the First (RE), Last (RE), Follow (RE, x), and Empty (RE) functions to build the NFA. The following is an example for RE = ((AT|GA((AG|AAA) *)). We first marks RE with order number as (A_1T_2|G_3A_4((A_5G_6|$A_7A_8A_9$) *)), the order number is the state number in the NFA. Then we mark the initial state by using First (RE), the First (RE) is the set of positions at which the reading can be started. For example: First (A_1T_2|G_3A_4((A_5G_6|$A_7A_8A_9$) *))= {1 ,3 }. After mask the initial state, we can marked the final state by using Last(RE), the Last(RE) is the set of positions at which a string can be recognized. For instant: Last (A_1T_2|G_3A_4((A_5G_6|$A_7A_8A_9$) *))={2 ,4 ,6 ,9 }. Then we compute the Follow (RE, x) for each state x, the Follow (RE, x) is the all positions in RE accessible from state x, for instance: Follow ((A_1T_2|G_3A_4((A_5G_6|$A_7A_8A_9$) *)),6)= {7,5}. This NFA is much smaller the Thompson NFA, and in the searching phrase, it only requires to store the related Follow (RE, x) if there are more than one next transition.

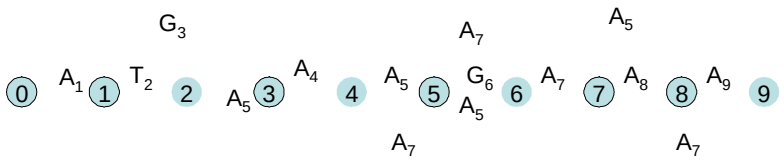


Fig. II.6. A Glushkov's construction example.

A.4 FILTER ALGORITHM

Filter algorithm directly compares the pattern with text character by characters. This kind of algorithm has some heuristic technique to save the comparison, the first

one is Knuth-Morris-Pratt(KMP)[22], it is a simple algorithm and utilize the prefix substring only. The second is Boyer–Moore[22] algorithm, which uses bad character and good suffix shift for a mismatch. And the third algorithm is counting filter which uses counting technique to find out the similar text portion with pattern.

A.4.1. KMP ALGORITHM

KMP is a single pattern algorithm and performs the left to right scan, it consists of two phases. In preprocessing phase it computes the prefix function for the pattern P. and in the searching phase, the pattern is searched against text with the prefix table. Table 1 is a prefix table for pattern ATCACATCATCA. Each cell of prefix row records the current suffix length that is also a prefix.

In the searching phrase, the pattern is compared against the text character by character. If a mismatch occurs, it have two cases for the following actions. Case 1 : when no prefix is found, it slides the pattern to right with the compared length. Case 2 : when a prefix is found at current suffix, it slides the pattern to right with compared length minus the prefix length. Table 2 is an example for the KMP string matching, when a mismatch occurs at position 3, ATC has no prefix, thus the row 4 is the case 1. The row 5 and row 6 are the case 2, because they have prefix A and ATC. Case 2 is shift less than case 1.

TABLE II.1. The KMP prefix table.

Pattern	A	T	C	A	C	A	T	C	A	T	C	A
Prefix	0	0	0	1	0	1	2	3	4	2	3	4

TABLE II.2. A KMP example.

Position	1	2	**3**	4	5	6	7	**8**	9	10	11	12	13	14	**15**	16	17	18	19	20	21	22

Text	A	T	**G**	A	T	C	A	**T**	C	A	C	A	T	C	**G**	T	A	A	A	A	A	A
Pattern	A	T	**C**	A	C	A	T	C	A	T	C											
Case 1				A	T	C	A	**C**	A	T	C	A	T	C								
Case 2							A	T	C	A	C	A	T	C	**A**	T	C					
Case 2												A	T	C	A	C	A	T	C	A	T	C

A.4.2. BOYER-MOORE ALGORITHM

This algorithm uses three techniques, which includes (1) scans from right to left in a pattern length window, (2) uses bad character shift rule and (3) computes last occurrence and good suffix function. In preprocessing phrase, it builds the bad character and good suffix shift table. The bad character shift table records the last occurrence position for each alphabet; Table II.3 is an example for pattern ATCACATCATCA.

TABLE II.3. The Boyer-Moore bad character table.

Alphabet	A	G	C	T
Bad char	12	0	11	10

Table II.4 is a good suffix table, it has two cases to compute the shift value, case 1 stores the rightmost position of a rightmost substring that is the right suffix substring at current position (not include the current character). Case 2 records the length of prefix is also a suffix at current suffix length from right to left. The good suffix shift is the pattern length m minus the maximum case 1 or case 2.

TABLE II.4. The Boyer-Moore good suffix table.

Good Suffix	1	2	3	4	5	6	7	8	9	10	11	12
Pattern	A	T	C	A	C	A	T	C	A	T	C	A
Case 1	0	0	0	0	0	0	9	0	0	6	1	11
Case 2	4	4	4	4	4	4	4	4	1	1	1	0
Good suffix	8	8	8	8	8	8	3	8	11	6	11	1

In searching phrase, when a mismatch occurs, it shift text by some shift amount, the shift amount = max(bad_char_shift, good_suffix_shift). It checks the bad character table and good suffix table. The left shift amount is the maximum value from these two tables. If always shift by one, its worst case searching time is O(nm). But shift by more than one is possible, and it has best case O(n/m). During the searching, the pattern windows with text by sliding from left to right, but in the pattern windows, pattern compares with text from right to left.

Table II.5 is an example for Boyer Moore string matching. When a mismatching occurs at “A” character in row 2, the bad character shift value is 2 and good suffix shift value is 1, thus it chooses the bad character shift, and shift to right by 2.

TABLE II.5. A Boyer-Moore string matching example.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Text	G	A	T	C	G	A	T	C	A	C	A	T	C	A	T	C	A	C	G	A	A	A	A	A
Pattern	A	T	C	A	C	A	T	C	A	T	**C**	A												
Bad char			A	T	C	A	C	A	T	C	A	T	**C**	A										
Good suffix					A	T	C	A	C	A	T	C	A	T	**C**	A								

A.4.3. COUNTING FILTER

Counting filter increases the matched counter when a character of the text window appears in patterns. And if the matched counter is equal to pattern's length that indicates the text windows might be matched with pattern, thus it verify it by a exact comparison. In the searching phrase, each alphabet c in pattern requires a counter A[c] to maintain the difference between pattern and text window, and a matched counter MA to indicate whether a matching might be occur.

In preprocessing phrase, A[c] is set to the number that c appears in the pattern and MA is set to zero. In searching phrase, we first process initial text windows, the MA=MA+1 if $A[t_j] > 0$ and $A[t_j]$ -1 after processing MA. Then for moving in a character, we MA=MA+1 if moving in $A[t_j] > 0$ and $A[t_j]$ -1 after processing moving in, and for moving out a character, we MA=MA-1 if moving out $A[t_j\text{-}m\text{-}k] > 0$ and $A[t_j\text{-}m\text{-}k]$ +1 after processing moving out. There might be a possible match if MA = length of pattern. Table II.6 is a example for search a pattern "aloha" over the text window "helloa". Initially, it set A[a]=2, A[l]=1, A[o]=1, A[h]=1, A[e]=0, MA = 0. And the result is that, in the row 6 and column 7, the MA=4 means that there might be a match, thus we can verified this text windows using the exact comparison.

TABLE II.6. A Counting Filter example.

Initial	A[a]=2	A[l]=1	A[o]=1	A[h]=1	A[e]=0	MA = 0
Read h	A[a]=2	A[l]=1	A[o]=1	A[h]=0	A[e]=0	MA = 1
Read e	A[a]=2	A[l]=1	A[o]=1	A[h]=0	A[e]=-1	MA = 1
Read l	A[a]=2	A[l]=0	A[o]=1	A[h]=0	A[e]=-1	MA = 2
Read l	A[a]=2	A[l]=-1	A[o]=1	A[h]=0	A[e]=-1	MA = 2
Read o	A[a]=2	A[l]=-1	A[o]=0	A[h]=0	A[e]=-1	MA = 3
Read a	A[a]=1	A[l]=-1	A[o]=0	A[h]=0	A[e]=-1	MA = 4

This filter algorithm can also be applied to multiple patterns with a slight modification. Multiple A[c] of multiple patterns can be packed into a computer words and update them in one iteration, and the multiple MA can be packet into a computer word as well.

A.5 BIT PARALLEL ALGORITHM

Bit Parallel algorithms use the bit state vector to simulate the nondeterministic finite automaton (NFA) in string matching, thus bit parallel algorithms are powerful as the automaton algorithms, more over the bit parallel are easier to handle and the class, allowing error and multiple pattern features of string matching. In bit parallel algorithm, each bit of the state vector represents a state in NFA. Bit parallel normally requires to build occurrence table for the patterns in preprocessing phrase, the occurrence table store the occurring bit vector of pattern for each alphabet. In the searching phrase, for each reading of character in text, it fetches the bit vector from occurrence table and does the bitwise operation with current state vector to produce a new state vector. The new state vector represents the current matched status. The shift-and is basic bit parallel algorithm, in its searching phrase, its updated function is $D' = ((D << 1 \mid 0^{m-1}1) \,\&\, B[t_j]$. Where D is current state mask, D' is next state vector, m is length of pattern. B is occurrence table, and t_j is current character of text.

In this section, we survey significant bit-parallel algorithms that include shift-or [25], advanced bit parallel methods [26] and bit parallel suffix automaton [27], and bit parallel regular expression [28]. The basic idea of shift-or is similar to shift-and but with different bitwise operation and different state vector representation. And the advanced bit parallel methods are showed by Sun-Manber that modified shift-or for

long pattern, approximation, allowing classes, multiple patterns. The algorithm simulates suffix automaton in bit parallel, and the NFA also can be simulated by a bit parallel method for the regular expression.

A.5.1. SHIFT-OR ALGORITHM

Shift-or has preprocessing and searching phrase. In the preprocessing phrase, it builds a pattern occurrence table $T[x]$. The row of T is the number of all possible alphabets that might appear in the text, and the column of T is the length of pattern. If an alphabet x appear in the corresponding position of pattern, then set $[x]$ bit to 0, otherwise to 1.

Fig. II.7 is a processing to construct the T. Firstly, the pattern "ababc" will be revere to "cbaba". And finding the $T[a]$, $T[b]$, $T[c]$ and $T[d]$, if the "cbaba" appear in the corresponding bit, set it to 0, otherwise to set to 1. For example $T[a]$ is in the bit 3 and bit 5 of pattern "cbaba", thus $T[a]$ = 11010.

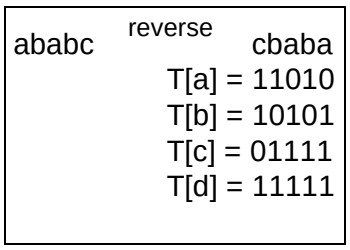


Fig. II.7. An Shift-or occurrence table example.

In the searching phrase, when a character t_i of text is inputted, a bit vector $State_i$ has a bitwise operation $State_i = (state_{i-1} << b) \mid T[t_i]$ to update the $State_i$, in the exact matching the shifted amount b is 1. When the leftmost bit of $State_i$ is zero that means a matching is found. Table II.7 is a searching example for pattern = ababc and text = abdababababc. In the last column of $State_i$ is 01111, the leftmost bit is 0, that

means it finds a matching.

TABLE II.7. A Shift-or example.

Text	**a**	**b**	**d**	**a**	**b**	**a**	**b**	**a**	**b**	**c**
$T[t_i]$	11010	10101	11111	11010	10101	11010	10101	11010	10101	01111
$State_{i-1}$ << b	11110	11100	11010	11110	11100	11010	10100	01010	10100	01010
$State_i$ ($State_{i-1}$ << b | $T[t_i]$)	11110	11101	11111	11110	11101	11010	10101	11010	10101	01111
<<	11100	11010	11110	11100	11010	10100	01010	10100	01010	

A.5.2. ADVANCED BIT PARALLEL ALGORITHMS

The advanced bit parallel can be applied to the shift-and or shift-or algorithms and that includes the long pattern, class and multiple patterns features. For handling the long pattern, we need to partitions pattern p into subpatterns p_i, and builds an array of D and B, process each part with basic algorithm, and if p_i is found than it process p_{i+1}. For handling the class, the class is that a character of pattern can represent many alphabets. This method needs to modify B table only, and have the ith bit set for all chars belonging to ith position in pattern. As for handling multiple pattern, we have two methods. The first way is that we can interleaves patterns and build the table in preprocessing phrase, and no need to modify the updated function. In the searching phrase, we shift r bit for each update for D and it just concatenate, where r is the number of patterns. The second method is that we shift 1 bit, but we need to modify the update function as $D = (D << 1) \,\&\, (1^{m-1}0)r$.

A.5.3. BIT PARALLEL FOR SUFFIX AUTOMATON ALGORITHM

This algorithm uses bit parallel to simulate the suffix automaton algorithm. It is able to handle class, multiple patterns, and allow errors, and faster than suffix automaton from 20% to 25%, and its updated function is $D' = D \& B[t_j] << 1$ which is similar to shift-and algorithm. Table II8 is an example for pattern = {aabbaab}, text = {abbabaabbaab}, D=1111111, B={{a,1100110},{b,0011001}}, m=7, last = 7, j=7.

TABLE II.8. An example of bit parallel for suffix automaton.

Text	D&B[tj]<<1	D'	j	Last
[abbaba(a)]bbaab	1111111 1100110	1100110	6	6
[abbab(aa)]bbaab	1001100 1100110	1000100	5	5
[abba(baa)]bbaab	0010000 1100110	0001000	4	5
[abb(abaa)]bbaab	0010000 1100110	0000000	3	5
Abbab[aabbaa(b)]	1111111 0011001	0011001	6	7
Abbab[aabba(ab)]	0110010 1100110	0100010	5	6

Abbab[aabb(aab)]	1000100 1100110	1000100	4	4
Abbab[aab(baab)]	0001000 0011001	0001000	3	4
Abbab[aa(bbaab)]	0010000 0011001	0010000	2	4
Abbab[a(abbaab)]	0100000 1100110	0100000	2	4
Abbab[(aabbaab)]	1000000 1100110	1000000	0	4

A.5.4. BIT PARALLEL FOR REGULAR EXPRESSION

Bit parallel can simulate Thompson and Glushkov NFA to matching the regular expression patterns. Thompson NFA bit parallel updated functions are $D = ((D << 1 \mid 0^{m-1}1) \,\&\, B[t_j]$ and $D' = E[D]$ functions. Where D is Sate mask, B is Occurrence Table, m is string length, t_j is current character, and $E[D]$ is null-closure function that is reachable state from D with null input. The Glushkov NFA Bit parallel has two updated function $T[D] = \bigcup_{i \in D} Follow(i)$ and $D' = T[D] \,\&\, B[t_j]$ in its searching phrase. T is a table that the states can be reached from an active state. The B and D are the same as the Thompson NFA bit parallel.

II.4 HARDWARE IMPLEMENTATION FOR STRING MATCHING

There are some existing hardware solutions for string matching algorithm. In our study, we classify them into the three categories; they include systolic array, parallel and pipeline hardware, and reconfigurable implementations.

B.1. SYSTOLIC ARRAY HARDWARE IMPLEMENTATION

Systolic array is suitable to implement the dynamic programming algorithm, the [29], [30], [31] and [32] has proposed the systolic array architecture for dynamic programming algorithms. Systolic array has the advantages that (1) does not need the preprocessing time and extra space (2) the search time is constant. However its drawbacks are that include (1) each cell systolic array require larger circuit than other algorithms' control circuits. (2) Its worst case is its best case, thus it is hard to improve the performance, (3) it cannot perform the string matching for wildcard or multiple patterns.

In the systolic array architecture of [30], the systolic array consists of $n*m$ processing element (PE), the n is length of text and m is the length of pattern. Fig II.8. depicts a PE architecture, each PE can determine edit distance cost from previous cost, that cost consists of PE's substitution cost *Sub=r(A(i)->B(j))*, deletion cost *Del=r(A(i)->Null)* and insertion cost *Ins=r(Null->B(j))*. The current PE's edit distance cost *D(i,j)* is the minimum value among *Sub + D(i-j,j-1)*，*Del +D(i-1,j)*，*Ins+D(i,j-1)*. Input *A(i)* and *B(j)* is the character at *i*，*j* index of the text and pattern, and input *D(i-j,j-1)*，*D(i-1,j)*，*D(i,j-1)* are the left-upper, left and upper edit distance cost. The final matched result is determined from the *D(i,j)* of the most right-bottom PE.

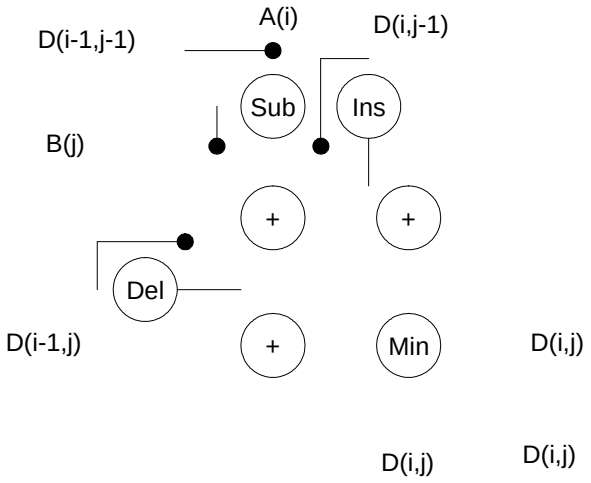


Fig. II.8. Processing element of dynamic programming systolic array.

B.2. PARALLEL AND PIPELINE HARDWARE IMPLEMENTATION

This kind of implementation is applied to the naïve string-matching algorithm, which means no need any heuristics string matching algorithm. This algorithm accelerates the string matching speed by utilizing hardware parallelism. The [33] and [34] propose the pipeline and parallel architecture for the naïve string matching. Fig. II.9 is a pipeline and parallel processing example that described in [33]. Firstly, the input Text: {*t*1, *t*2, *t*3, *t*4, *t*5…} is shifted into the shift register S, to compared with character P1, P2, P3, P4 of the Pattern by using comparator Cp. Then the final matching result is accumulated by the and gates.

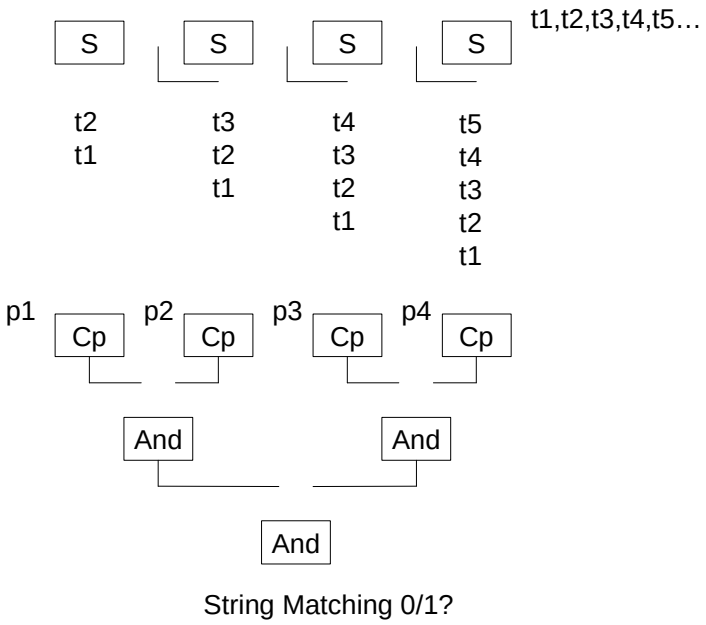


Fig. II.9. Pipeline and parallel processing example.

This kind of hardware acceleration has the merits that includes (1) no need the preprocessing on pattern. (2) Simple algorithm that compared with other heuristic algorithm, however, its drawbacks are (1) it pays the large circuit for parallelism and (2) long delay for long patterns.

B.3. RECONFIGURABLE HARDWARE IMPLEMENTATION

This hardware implementation hardwired the string matching circuit and its patterns directly into the reconfigurable hardware. The [35] and [36] demonstrated the implementation of regular expression string matching based on the reconfigurable FPGA architecture. Fig. II.10 is the technique used in [35], its regular expression pattern is a(b|c)* and it is hardwired in the reconfigurable circuit in the preprocessing phrase. In the searching phrase, the matching processing is that the characters of text is shifted into the circuit cycle by cycle. After the input is done the final matching result is reported after the few delayed cycles. The reconfigurable has the advantage in (1) no need the preprocessing table, and the compact circuit for the small patterns (2) more flexible than ASIC solution; however the reconfigurable hardware is much

expensive and slower than ASIC solution.

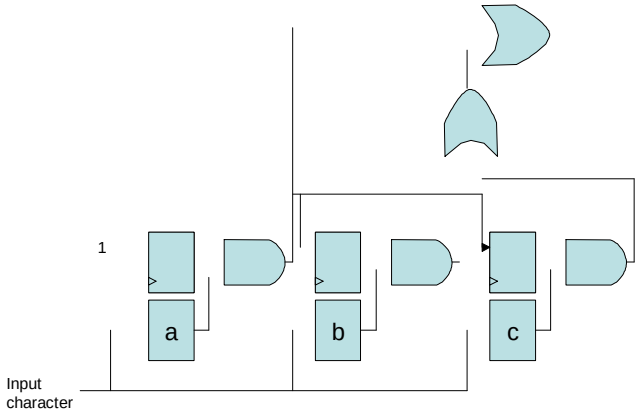


Fig. II.10. Reconfigurable hardware acceleration example

II.5 DISCUSSION

In this section, we discuss the requirements, and analyze the architecture for string matching algorithm, then we conclude a suitable algorithm and its hardware architecture for string matching.

C.1. STRING MATCHING REQUIREMENTS

The functionalities of the string matching have the exact matching, class, approximate, multiple patterns, wildcard, and regular expression string matching. In the Snort 1.9, its rule for string matching has the following fields.

- Content: search for a pattern in the packet's payload.
- Content-list: search for a set of patterns in the packet's payload.
- Offset: modifier for the content option, sets the offset to begin attempting a pattern match.
- Depth: modifier for the content option, sets the maximum search depth for a pattern match attempt.
- Nocase: match the preceding content string with case insensitivity.
- Regex: wildcard pattern matching, this is not normal regular expression.
- Uricontent: search for a pattern in the URI portion of a packet.

Based on snort and our observation, thus, we can summary the requirements of string matching as table 9. And we conclude the approximate, class, regular expression and long pattern is not mandatory in string matching, but the multiple patterns, wildcard, real-time and case sensitive are mandatory.

TABLE II.9. String matching requirements.

Functions	Description	Requirement
Allow Error	Allow k number of character error	Maybe
Multiple Pattern	Multiple pattern matching in one iteration	Yes
Class	One character represents multiple alphabet	Maybe
Wildcard	Don’t care of multiple characters	Yes
Regular Expression	Kleene Star, Concatenation, Or	Maybe
Long Pattern	Support long pattern	Maybe
Real Time	On fly processing	Yes
Case Sensitive	Alphabet is case insensitive	Yes

C.2. Recommend Hardware Implementation of String Matching

Fig. II.11 illustrates the hardware structure for four main string matching algorithms; (a) Bit parallel algorithm has an occurrence table to store the preprocessing pattern data, and its control circuit is simplest among other algorithm, it reads the text and table data to does the bitwise operation in the searching phrase. (b) Filter algorithm, its control circuit is more complicated than bit parallel, because it requires reading the good suffix shift table, bad character shift table, and computing the shift amount for each input character in text. (c) Dynamic programming algorithm, it has no central control circuit that is distributed in PE circuit as a systolic array. This array grows as the pattern or text length, thus it has larger circuit size than

other algorithms normally. (d) Automaton Aho-Corasick algorithm, its control circuit is more slower and larger than bit parallel algorithm, because it requires to reads goto table, failure table, output table, then does the byte comparison in order to determine next state and matched status.

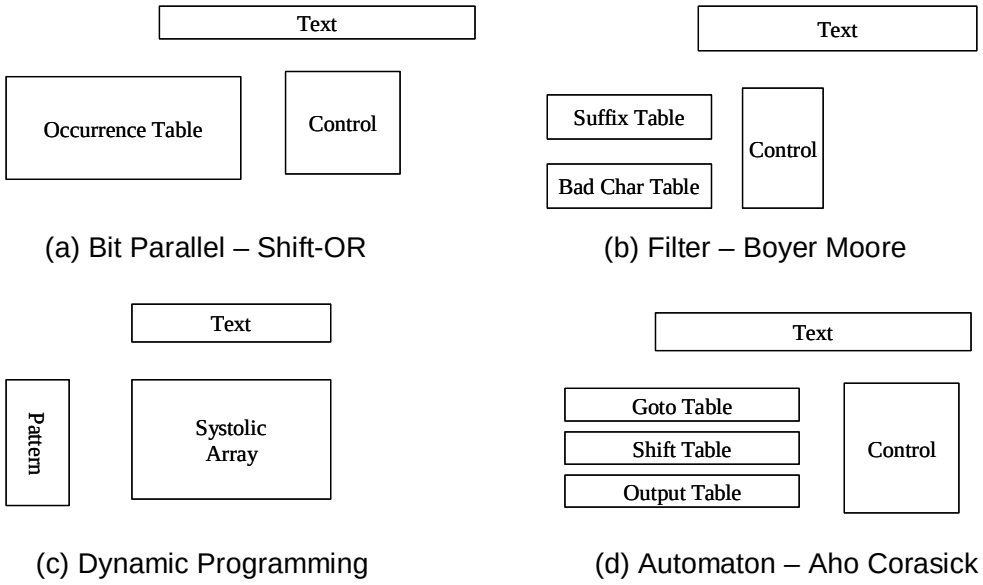


Fig. II.11. Four main architecture of string matching algorithms.

Which string matching algorithm is suitable to be implemented in hardware? We can consider them from three respects-capability, efficiency and implementation cost. Table II.10 is a summary of our observation; we summarize the advantage and disadvantage for string matching algorithms of the hardware implementations. In our final recommendation, the bit parallel algorithm is recommended for hardware implementation, because it is a simple and powerful string matching algorithm, it is able to some heuristic techniques as well.

TABLE II.10. Analysis of hardware Implementations of string matching algorithms.

Taxonomy	Advantage	Disadvantage	Recommendation
Filter	● Optimal average case in time complexity	● Not powerful as the automaton or bit parallel, support no multiple pattern, and allow error ● Need the adder and subractor in each shifting calculation	Yes (Weak, no multiple pattern and need more circuit than bit parallel)
Dynamic Programming	● No Preprocessing	● Less powerful, no support multiple pattern and long pattern. ● Each systolic array cell contain many adder and subractor.	No (suitable for single and short pattern)
Bit-Parallel	● Good worst case in time complexity ● Bitwise operation in each character comparison	● Powerful string matching functionalities, the similar to automaton. ● Pattern length is limited by bit mask length	Yes (Strong, simple architecture suitable for hardware implementation)
Automaton	● Good worst case in time complexity ● Flexible in pattern length	● Powerful string matching functionalities ● Full character comparison for each character comparison ● Table size is larger than Bit-Parallel	Yes (Weak, because it need more storage and lookup time then bit parallel)

II.6 CONCLUSION

We have a comprehensive survey over the existing the string matching algorithms and hardware implementations. Based on our observation, we have identified four type string-matching algorithm that includes dynamic programming, bit parallel, filter and automaton. The existing string matching hardwares focus on the systolic array, parallel architecture, and reconfigurable mostly. But these are not mandatory features that requires in the string matching hardware. String matching hardware requires the high speed, low cost and powerful functionalities.

In our analysis hardware implementation of four main types of string matching algorithms, the bit parallel algorithm is strong recommended for the hardware implementation, because it has the powerful string-matching feature as automaton and simplest control circuit for hardware implementation.

REFERENCE

[1] T. Chiueh and P. Pradhan, "High Performance IP Routing Table Lookup Using CPU Caching," in *Proc. of IEEE INFOCOM*, Apr. 1999.

[2] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," in *Proc. of ACM SIGCOMM'97*, 1997.

[3] V. Srinivasan and G. Varghese, "Fast IP Lookups Using Controlled Prefix Expansion," *ACM Trans. on Computer Systems*, vol. 17, pp. 1-40, Feb. 1999.

[4] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Trans. on Networking*, vol. 7, no. 3, pp. 324-334, June 1999.

[5] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, pp. 1083-1092, June 1999.

[6] S. Sikka and G. Varghese, "Memory-Efficient State Lookups with Fast Updates," in *Proc. of SIGCOMM*, 2000.

[7] P. Gupta, S. Lin, and N. Mckeown, "Routing Lookups in hardware at Memory Access Speeds," in *Proc. of INFOCOM*, 1998.

[8] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High-Speed IP Routing Table Lookups," in *Proc. of ACM SIGCOMM*, Sep. 1997.

[9] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High-Speed Prefix Matching," *ACM Trans. on Computer Systems*, vol. 19, pp. 440-482, Nov. 2001.

[10] M. A. Ruiz-Sanchez, et al., "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network Magazine*, pp. 8-23, Mar. 2001.

[11] M. Waldvogel, "Multi-Dimensional Prefix Matching Using Line Search," in *Proc. of IEEE LCN*, Nov. 2000.

[12] P. Warkhede, S. Suri, and G. Varghese, "Fast Packet Classification for Two-Dimensional Conflict-Free Filters," in *Proc. of IEEE INFOCOM*, 2001.

[13] Merit Networks Inc., Internet Performance Measurement and Analysis (IPMA) statistics and daily reports, in IPMA project, http://www.merit.edu/ipma/routing_table/

[14] http://www.mcvax.org/~jhma/routing/

[15] P.A. Hall and G.R. Dowling, "Approximate string matching", *ACM Computing Surveys*, vol. 12, pp. 381-402, 1980.

[16] G. Navarro, "A guided tour to approximate string matching", *ACM Computing Surveys*, vol. 33, pp.31-88, 2001.

[17] P. Sellers, "The theory and computation of evolutionary distances: pattern recognition", *Journal of Algorithms*, vol. 1, pp. 359-373, 1980.

[18] V. A. Alfred and J. C. Margaret, "Efficient string matching: an aid to bibliographic search", *Communications of the ACM*, vol. 18, June 1975.

[19] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Speeding up two string matching algorithms", *Algorithmica*, vol. 12, pp. 247-267, 1994.

[20] M. Raffinot, "On the multi backward dawg matching algorithm (MultiBDM)", in *Proceedings of the 4rd South American Workshop on String Processing*, pp. 149-165, Valparaiso, Chile, Nov., 1997.

[21] K. Thompson, "Regular expression search algorithm", *CACM*, vol. 11,

pp. 419-422, 1968.

[22] D. E. Knuth, J. H. Morris, and V. R. Pratt, “Fast pattern matching in strings”, *SIAM Journal of Computing*, vol. 6, pp. 323--350, June 1977.

[23] R. S. Boyer and J. S. Moore, “A fast string searching algorithm”, *Communications of the ACM*, vol. 20, Oct. 1977.

[24] Josué Kuri and Gonzalo Navarro, ”Fast Multipattern Search Algorithms for Intrusion Detection”, *SPIRE 2000*, pp. 169-180, 2000.

[25] R. A. Baeza-Yates and G. H. Gonnet, “A new approach to text searching”, in *Proceedings of the 12th annual international ACM SIGIR confe rence on Research and development in information retrieval*, vol. 23, May 1989.

[26] S. Wu and U. Manber, “Fast text searching: allowing errors”, *Communications of the ACM*, vol. 35, Oct. 1992.

[27] G. Navarro and M. Raffinot. “A Bit-parallel approach to Suffix Automata: Fast Extended String Matching”, in *Proc. CPM'98*, *LNCS 1448*, pp. 14-33, 1998.

[28] G. Navarro and M. Raffinot. “Compact DFA representation for fast regular expression search”, in *Proceedings of the 5th Workshop on Algorithm Engineering*, *LNCS 2141*, pp. 1-12, 2001.

[29] H.-M. Bluthgen and T.G. Noll, “A programmable processor for approximate string matching with high throughput rate”, in *Proc. of IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pp. 309-316, 2000.

[30] N. Ranganathan, K. Remedios, and R. Sastry, “CASM: a VLSI chip for approximate string matching”, in *IEEE Transactions on Pattern*

Analysis and Machine Intelligence, vol. 17, pp. 824-830, Aug 1995.

[31] Raghu Sastry and N. Ranganathan, "A Systolic Array for Approximate String Matching", in *Proc. of ICCD 1993*, pp. 402-405, 1993.

[32] N. Ranganathan and R. Motamarri, "A VLSI architecture for computing the optimal correspondence of string subsequences", in *Proc. of CAMP 1997*, Como, ITALY,Oct. 1997.

[33] K.M. George and H. P. Jin, "Parallel string matching algorithms based on dataflow", System Sciences, in Proc. of the 32nd Annual Hawaii International Conference, 1999.

[34] P. Moisset, P. C. Diniz, and J. Park, "Matching and searching analysis for parallel hardware implementation on FPGAs", *FPGA 2001*, pp. 125-133, 2001.

[35] R. Franklin, D. Carver, and B. L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware", *FCCM'02*, 2002.

[36] Reetinder Sidhu and Viktor K. Prasanna, "Fast Regular Expression Matching using FPGAs", in *IEEE Symposium on Field-Programmable Custom Computing Machines* (*FCCM 2001*), April 2001.