Dual Degree Dissertation

Finite State Machine based design for Packet Forwarding Engine

submitted in partial fulfillment of the
requirements for the degree of

Bachelor of technology
and
Master of Technology
(under the Dual Degree programme)

by
Kshitiz Saxena
Roll no. 96D07006

under the guidance of
Prof. Abhay Karandikar
and
Prof. Madhav P. Desai

Department of Electrical Engineering
Indian Institute of Technology
Powai, Mumbai-400 076
June-2001

Disseration Approval Sheet

The Dual Degree Project report entitled " **Finite State Machine based design for Packet Forwarding Engine** " submitted by **Kshitiz Saxena** (Roll No. $96D07006$) may be accepted.

Chairman: ——————

External Examiner: ——————

Guides: ——————

Internal Examiner: ——————

. ——————

Acknowledgement

I would like to express my sincere gratitude to **Prof. Abhay Karandikar** for his invaluable guidance, support and motivation during the course of this project. I thank **Prof. Madhav P. Desai**, who has been source of inspiration, for his invaluable suggestions and discussions during the course of project. I would also like to thank Shesh Murthy, Mukesh K. Mishra and Ritu Gupta who had helped me at various stages of my project. I would also like to thank my dual degree batchmates for giving me encouragement and making my stay at IIT memorable.

Kshitiz Saxena
(96D07006)

Abstract

Over the past few years, the Internet has been growing tremendously, largely because access and management of the Internet is far more cost effective. The continuous growth of the Internet has made it necessary to implement newer and innovative solutions to improve routing performance. The Packet Forwarding Engine is the prime point of control in determining the manner in which packets are handled within a router. The packet forwarding engine, can therefore, become a bottleneck. We have developed a novel architecture for a Packet Forwarding Engine which has salient features like flexibility, reusabillity and short design time. This architecture also has gate level implementation so that it will keep pace with ever growing processor technology.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A router consists of two major architectural components (Figure 1.1), *ie*, Packet Forwarding Engine and Routing Engine. The Packet Forwarding Engine is responsible for making all packet forwarding decisions, *ie*, route lookup, packet classification, buffer management and scheduling while the Routing Engine handles all the routing protocol processes, as well as other software processes that control the router's interfaces, a few of the chassis components, system management, and user access to the router.

All data packets traverse the path through the Packet Forwarding Engine where their headers are processed and forwarding decisions are made while all control packets from the network are directed to the Routing Engine and are responsible for modifying databases for route lookup and packet classification. The Packet Forwarding Engine and the Routing Engine perform their primary tasks independently, though they constantly communicate with each other. This arrangement provides streamlined forwarding and routing control and the capability to run backbone networks at high speeds.

The performance of the Packet Forwarding Engine decides the performance of router because all data packets have to pass through this component. The packet forwarding engine can therefore become a bottleneck. Thus an optimal solution for high speed router is a high speed forwarding engine which is the focal point of this report.

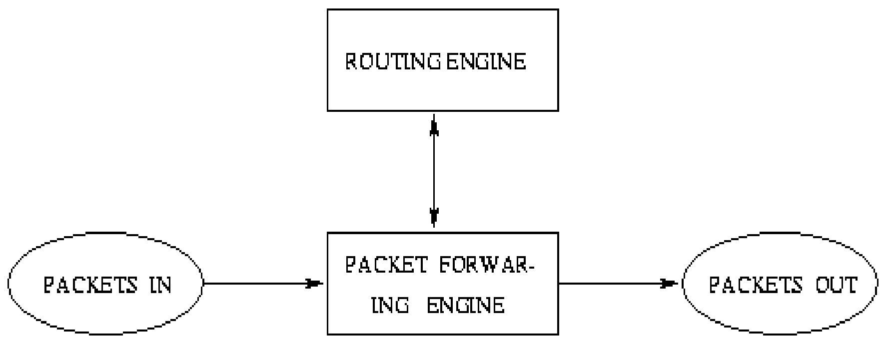


Figure 1.1: Component of Router

1.1 Role of Packet Forwarding Engine in High Speed Router

The Packet Forwarding Engine is the prime point of control in determining the manner in which packets are handled within the router. Forwarding engines must be able to identify the context of packets and apply the necessary actions so as to satisfy the user requirements. Such actions constitute: directing packets to appropriate output ports, classifying packets into different categories or special queueing and scheduling actions based on certain criteria. The Packet Forwarding Engine buffer incoming data packets according to its buffer management policies, does route lookup and packet classification according to rules specified and then schedule the packets onto the output link.

The maximum delay for header processing must not exceed the delay that the packet from the service class with least delay can experience. The forwarding engines must be able to process every packet arriving to the physical interfaces at wire speed. Processing packets at wire speed of Gigabits per second requires that all the units doing route lookup, packet classification, buffer management and scheduling must process millions of packets per second.

Achieving speed of processing millions of packets per second puts a severe constraint on the design procedures of forwarding engine. In the case of route lookup and packet classification algorithm, high speed can be obtained at the cost of larger memory usage, *ie*, trade-off is between memory usage and speed [1]. The speed of implementing buffer management and scheduling policies depends on their complexities. To improve speed we can employ a cluster of Packet Forwarding Engines which work either in dedicated or in polled fashion to increase the throughput, but this solution is not cost effective. The cost effective solution is to come up with an optimized design for the Packet Forwarding Engine. Mainly optimization is required in the area of route lookup and packet classification as they are major bottlenecks in high performance routers and are discussed in detail in the following chapter.

The report is organized in the following format. We will look at the problem of Route Lookup and Packet Classification and present some approaches to solve these problems in Chapter 2. Our novel architecture for the Packet Forwarding Engine is presented in Chapter 3. We begin with using *One Bit Trie* algorithm for generating Finite State Machines (FSMs) for the above mentioned problems of Route Lookup and Packet Classification. This is explained in detail in Chapter 4. Chapter 5 deals with the generation of FSMs using *Recursive Flow Classification* algorithm. A new technique to generate FSM for route lookup is discussed in Chapter 6. The characteristic of one bit trie FSM for route lookup generated for different databases are presented in Chapter 7. The topographical breakdown of one bit trie FSM is explained in Chapter 8. The results of FSM implementation in VHDL are presented in Chapter 9. Concluding remarks and future work are presented in Chapter 10.

Chapter 2

Route Lookup and Packet Classification

We will begin with the specification of Route Lookup problem followed by Packet Classification and then present several existing solutions for these problems in brief.

2.1 Route Lookup

Route Lookup is a major bottleneck in the high performance router. It is a challenging problem because it requires a longest matching prefix lookup[1]. The increasing routing table sizes and traffic, and migration from IPv4 (32 bits) to IPv6 (128 bits), has further increased the complexity.

2.1.1 Problem Statement

Forwarding is a decision to which interface a given packet should be forwarded. It is based on packet's Destination Address (DA) field. The router has to perform this operation for every packet. Route lookup would have been simple if we could lookup an IP destination address in a table that lists the output link for each assigned Internet address. In this case, lookup could be done by hashing, but a router would have to keep millions of entries (for IPv4 2^{32}entries in the worst case). To reduce database size and routing update traffic, a router database consist of a smaller set of prefixes. This reduces the database size, but at the cost of requiring a more complex lookup called *longest matching prefix*. It also requires a more complex update procedure when prefixes are added or deleted. Each Internet router can have a potentially different set of prefixes, each of which we will denote by a string (e.g. 01*) of up to 32 bits followed by "*". Thus if the destination address begins with 01000 and we have two prefix entries (01*→L1 ; 0100*→L2), the packet should be switched to link L2.

2.1.2 Techniques employed for Route Lookup

We give a brief description of various techniques used for route lookup :

1. **Hardware solution** [1]: Content Addressable Memories (CAMs) are the perfect solution for a really small search space. These special-purpose chips are fast and reliable . However, this solution is very expensive and are difficult to scale.

2. **Routing Lookup in Hardware at Memory access Speeds** [2]: This algorithm takes advantage of common lengths of prefixes. Today memory is relatively cheap, and with some

intelligent hierarchy, very high speeds can be achieved with moderate memory requirements. However, these algorithms are only a temporary solution, because direct lookup will not be feasible for IPv6 addresses. Insertion and removal can be very expensive, especially in the case of a short prefix.

3. **Fast Address Lookups using Controlled Prefix Expansion** [1]: The basic idea is to reduce a set of arbitrary length of prefixes to a predefined set of lengths using a technique called controlled prefix expansion. If the longest prefix is of length W and if they are expanded to K distinct lengths then linear search time changes from O(W) to O(K). The K distinct lengths are chosen by dynamic programming such that memory utilization is minimized.

4. **Deterministic IP Table Lookup at Wire Speed** [3]: In this approach physical memory is divided into segments, which can be uniquely addressed. Each of those segments has an identical number of words, and each word within any given segment can be identified by an offset. Destination address (DA) is divided into equal length substrings. Using these substrings access to subsequent segments are made. Output of last matching prefix is always kept stored. Indexing into the memory segments using substrings is continued till leaf node is reached. If no match is found at leaf node, the port number corresponding to last matching prefix is out-putted.

The last two techniques explained above seems to be promising in development of high speed route lookup. As mentioned earlier trade off is between memory usage and speed. Since the database entries changes at very high rate (especially in case of core router) thus selection of algorithm is dependent on the time required to update.

2.2 Packet Classification

Routers classify packets to various classes or flows that can be provided differentiated qualities of service. Classification can be based on arbitrary number of fields in the packet header. Performing packet classification quickly on an arbitrary number of fields is known to be difficult and has poor worst case performance [5].

2.2.1 Problem Statement

A general filter consists of arbitrary prefix or range specifications on different header fields of the incoming packet, like source address, destination address, protocols, port numbers etc. The problem of packet classification is defined as follows. Suppose there are K header fields in each packet that are relevant to filtering. A filter database consists of N filters $F_1....F_N$. Each filter F_i is a K tuple (F_i[1],....,F_i[K]), with each F_i[j] being a variable length prefix bit string. Each field *j* in a filter is allowed three kinds of matches : exact, prefix or range. We say that a packet P matches a filter F_i if for all packet fields *j*, F_i[j] matches P[j]. The problem of packet classification can be viewed as one of mapping K bits in the packet header to T bits of Class ID (where, T = log(N) and T$\ll$K).

2.2.2 Techniques employed for Packet Classification

We give brief description of some of the techniques used for Packet Classification.

1. **Hardware Solution** [4]: These schemes use CAMs for exact matching. Different CAMs works in parallel to compare the input key value to the content of memory location. This can be generalized if the CAMs allow certain field positions to be masked out. However, it is difficult to manufacture CAMs with the width required to solve filter problems. Even hardware solutions suffer from the problem of inflexibility.

2. **Tuple Space Search**[4]: This is based on the observation that number of distinct prefix lengths tends to be small and so will be the case with combinations of prefix lengths. Here a tuple is defined for each combination of field length and the resulting set is called tuple space. Suppose the filters are K-dimensional filters. A tuple T is a vector of K lengths. We say a filter F belongs or maps to tuples T if the i_{th} field of F is specified to exactly T[i] bits. For example both F_1=(01*,111*) and F_2=(10*,110*) map to tuple [2,3]. The required number of bits from each filters are concatenated to construct a hash key for that filter. All the filters mapped to T are stored in a hash table Hash-table(T). Probing is done by concatenating the required number of bits from the packet as specified by T and then doing a hash in Hash-table(T). In linear search also the worst case time will be proportional to the number of distinct tuples which will be less than the number of filters. The further refinement of Tuple Space Search is discussed in [4].

3. **Recursive Flow Classification** [5]: The Recursive Flow Classification algorithm has several phases, each phase consisting of a set of parallel memory lookups. In the first phase, F fields of the packet header are split up into multiple chunks that are used to index into multiple memories in parallel. The contents of each memory are chosen so that the result of the lookup is narrower than the index, *ie*, fewer bits. In subsequent phases, the index into each memory is found by combining the results of earlier phases. In the final phase we are left with one result from the lookup which correspond to the class to which the packet belongs. This scheme can reduce storage requirement by several orders and provide good searching speed.

To provide differentiated services we require packet classification and most importantly it should be done at wire speed. The last two schemes seems to be promising. The choice of algorithm will be dependent on the memory available and speed requirements.

Chapter 3

Proposed Approach

The hardware solutions doing specific jobs are fast and reliable but hardware design is relatively difficult, time consuming, inflexible and specialized relative to software design. Hardware designs are not reusable as compared to software designs and significant efforts are required to transfer them to new generation. Thus if we come up with an architecture which combines the features of hardware and software solutions then we can find optimized solutions to both of the above problems. The above idea is the basic motivation behind our new approach.

What is needed is a system approach that combines the high performance associated with hardware with short design time, flexibility and reusability of software. We divide the problem of Route Lookup and Packet Classification into smaller parts which can be separately implemented in hardware and software. Figure 3.1 gives a view of proposed architecture.

In the architecture shown in the Figure 3.1, there is a dedicated hardware component, which does the route lookup or packet classification. The hardware implementation is in the form of Finite State Machine. The hardware component is designed to support some maximum number of states which is greater than the number of states possible. The hardware component is programmable like Field Programmable Gate Arrays and the state transition table can be loaded onto it by the processor. The number of states used changes as per the requirement.

The Processor implements all the algorithms for route lookup and packet classification and then compiles it to the format which can be loaded onto the programmable hardware. The Processor is responsible for the generation of Finite State Machine. It takes data from the Database which is maintained by the Routing Engine and generates Finite State Machine according to the algorithm being implemented. It is also responsible for changing the finite state machine loaded onto the programmable hardware if some entries in the database change. In case of changes either the entire state machine is generated again and loaded onto the programmable hardware or only required changes are done in the finite state machine.

This architecture is flexible as any software changes can be accommodated in the Processor and the modification of hardware component is only required when there is technological change. Thus this architecture has all the required features, *ie*, short design time, flexibility and reusability.

All the previous solutions for route lookup or packet classification required several memory accesses. Thus the performance was largely dependent on speed at which memory can be accessed. Even though there has been considerable technological advancement in field of memory design but it had not kept pace with the advancement in processor technology. Thus our architecture has an additional advantage as it has gate level implementation. The FSM based hardware component can be very efficiently implemented using Flip Flops and all the memory accesses can be reduced to accessing the high speed registers.

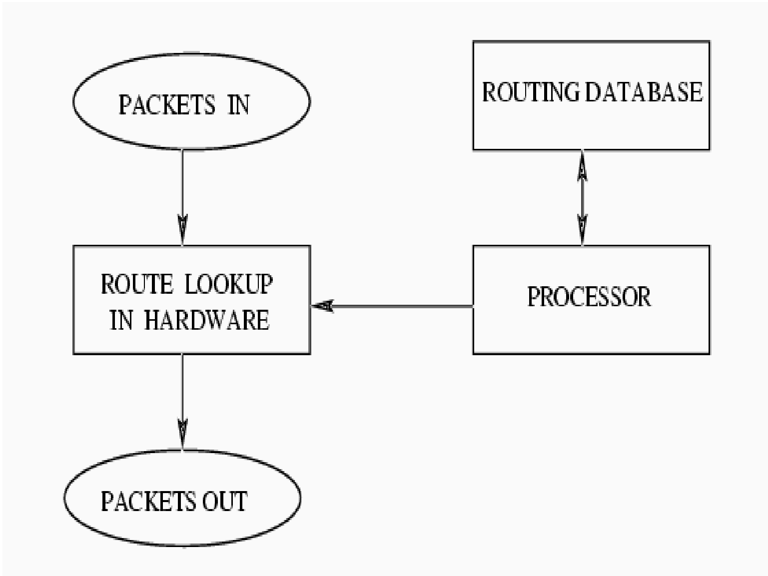


Figure 3.1: Proposed Architecture

Now, the first step is to develop Finite State Machines for Route Lookup and Packet Classification. We begin with very simple algorithms like one bit trie for the above problems and generate Finite State Machines for databases available for them, and then, other sophisticated algorithms like recursive flow classification will be implemented to optimize the number of states required.

Chapter 4

Finite State Machine for Route Lookup and Packet Classification

We begin our first step with the development of Finite State Machines for Route Lookup and Packet Classification. As a first step, we start with an assumption that databases are static and if they change then finite state machine is generated again from the beginning.

4.1 Route Lookup

The Route Lookup problem is to find the best matching prefix. We assume the destination address is inputted in serial order. We start at initial node and state transition is done using these bits inputted one after another until it reach the final state. We always keep track of the last matching prefix and it is reported after final state is reached. Since input is bit by bit serial, this finite state machine is similar to the one bit trie. Our state stores the following information, state number, info (if its prefix matches some entry in database then it stores the port number corresponding to it otherwise it is NULL), prefix (all the destination address reaching that state will contain that prefix), and links to two states one having prefix prefix+'0' (or final) and other having prefix prefix+'1' (or final) which are next states reached after transition depending on whether the input bit is '0' or '1'. The code which generates finite state machine similar to one bit trie is as follows.

```
begin from node 'initial'
initial→prefix='' // initial state does have any prefix
make a link list pointing to node 'initial'
while( link list is not empty)
{
    create new empty link list
    while ( end of link list is not reached )
    {
        prefix←prefix of the node at the head of link list

        if prefix+'0' exist in database then
            create a node with that prefix and
            put a link to that node in node→zero and
            put that node in new link list
        else
```

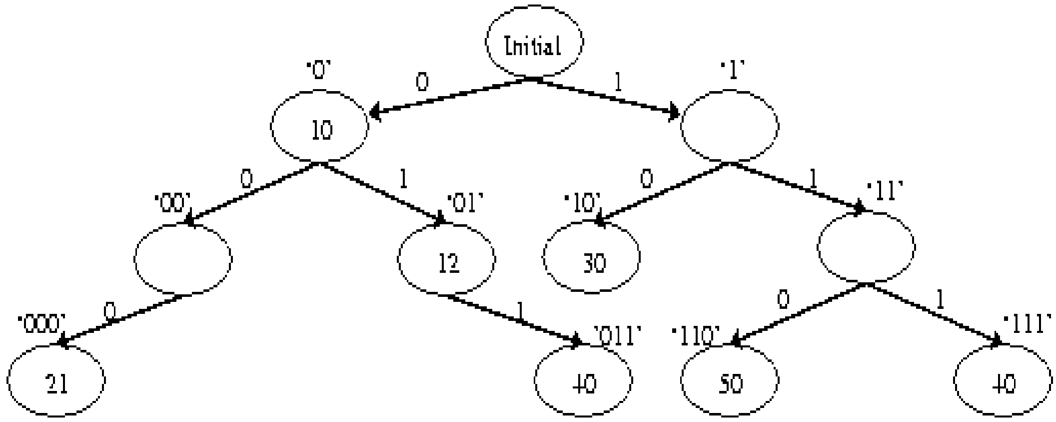


Figure 4.1: One Bit Trie corresponding to given database

```
            link node→zero to final

        if prefix+'1' exist in database then
            create a node with that prefix and
            put a link to that node in node→one and
            put that node in new link list
        else
            link node→one to final

        move to next node in the link
    }
    move new link list into link list
}
```

The output of above code is a link list structure corresponding to the given database. When the database contains the following entries (format is prefix !port number) 0*→10, 01*→12, 011*→20, 000*→21,10*→30, 111*→40, 110*→50, Figure 4.1, illustrates one bit trie for this database. The nodes of one bit trie either have information about port number or are empty. Figure 4.2, illustrates the state transition diagram which is very much similar to one bit trie structure. The state transition diagram illustrates different states in the FSM and are connected by links. Each link illustrates bit required for that transition and the corresponding output.

The output of the code in form of transition table is given in Table 4.1. The State Transition table gives all details of the Finite State Machine. It states all the states and transitions in the FSM. It specifies the next state reached from the current state depending on the input which is bit '0' or '1' in this case. The output corresponding to each transition, which is the port number if the next state prefix matches some entry in database otherwise NULL, is mentioned.

4.2 Packet Classification

The packet classification is much more complicated problem than route lookup. Here instead of one field, *ie*, destination address, the matching is done using multiple fields like source address,

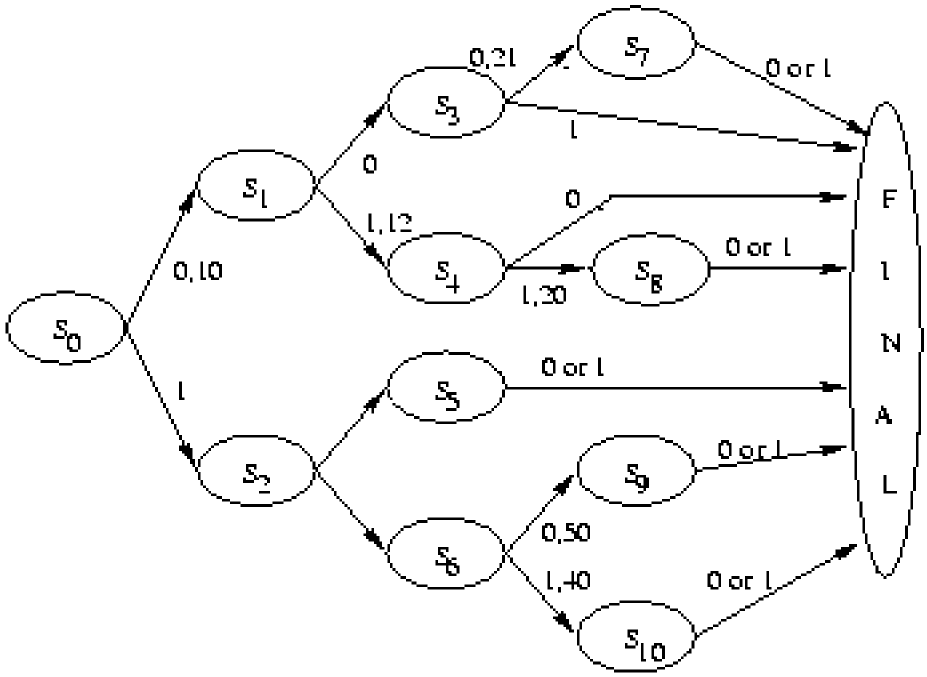


Figure 4.2: Finite State Machine corresponding to given database.

Initial State	Prefix	Bit Input ’0’ Next State	’0’ Output	Bit Input ’1’ Next State	’1’ Output
S_0	‘’	S_1	10	S_2	
S_1	‘0’	S_3		S_4	12
S_2	‘1’	S_5	30	S_6	
S_3	‘00’	S_7	21	Final	
S_4	‘01’	Final		S_8	20
S_5	‘10’	Final		Final	
S_6	‘11’	S_9	50	S_{10}	40
S_7	‘000’	Final		Final	
S_8	‘011’	Final		Final	
S_9	‘110’	Final		Final	
S_{10}	‘111’	Final		Final	
Final		Final		Final	

Table 4.1: State Transition Table

destination address, port numbers etc. There is also an issue of conflicts when one header can match more than one rule but for time being that case is ignored and will be investigated later.

We have designed our finite state generation code for the following format of rule: source address, mask / destination address, mask / range of input port number / range of destination port number. As an example the rule will be given as 144.0.0.0, 255.0.0.0 / 192.168.0.0, 255.255.0.0 / 16-250 / 255-2047. The mask corresponding to source and destination address are bitwise anded with source and destination address of the header and then compared with them. The use of mask allows number of addresses to map onto same rule. Thus in above case all packet header with source address 144.*.*.* and destination address 192.168.*.* will match first two fields of above mentioned rule. For last two fields we have to do range matching.

We have implemented Packet Classification problem also as *longest matching prefix*. Thus first step was to develop prefix table from given database and then develop a FSM using code for route lookup. The optimum number of prefixes matching the rule are generated. As an example we present the optimum number of prefixes for above rule.

x implies *don't care bit*

$$\text{144.0.0.0, 255.0.0.0}\rightarrow\underbrace{10010000}_{144}.\underbrace{xxxxxxxx}_{0}.\underbrace{xxxxxxxx}_{0}.\underbrace{xxxxxxxx}_{0}$$

$$\text{192.168.0.0, 255.255.0.0}\rightarrow\underbrace{11000000}_{192}.\underbrace{10101000}_{168}.\underbrace{xxxxxxxx}_{0}.\underbrace{xxxxxxxx}_{0}$$

$$\text{16-250}\rightarrow\begin{cases} 0001xxxx & 16-31 \\ 001xxxxx & 32-63 \\ 01xxxxxx & 64-127 \\ 10xxxxxx & 128-191 \\ 110xxxxx & 192-223 \\ 1110xxxx & 224-239 \\ 11110xxx & 240-247 \\ 1111100x & 248-249 \\ 11111010 & 250 \end{cases}$$

$$\text{256-2047}\rightarrow\begin{cases} 00000001xxxxxxxx & 256-511 \\ 0000001xxxxxxxxx & 512-1023 \\ 000001xxxxxxxxxx & 1023-2047 \end{cases}$$

The prefix table is made by concatenating prefixes of different fields. Thus corresponding to above rule, we have total of 9x3=27 prefixes. There is one interesting thing in these prefixes, it contains don't care bit and if these appear towards the end of the prefix then we can neglect them while making entry into the prefix table. As an example we illustrate one entry in the table :

144.0.0.0, 255.0.0.0 / 192.168.0.0, 255.255.0.0 / 16-31 / 255-1023

$$\underbrace{10010000xxxxxxxxxxxxxxxxxxxxxxxx}_{144.0.0.0,255.0.0.0}\ \underbrace{1100000010101000xxxxxxxxxxxxxxxx}_{192.168.0.0,255.255.0.0}$$
$$\underbrace{0001xxxx}_{16-31}\ \underbrace{00000001xxxxxxxx}_{255-1023}$$

Now we move to next step, *ie*, generation of Finite State Machine from the prefix table. The algorithm is same as in case of route lookup except for two things. First, if don't care bits exist at the same place in all the prefixes then we can optimize the number of states required by skipping those bits. Second, while comparing prefixes with entries in the prefix table, all don't care bits must be skipped. Generally the state transition table generated even for small number of rules are large enough and thus an example is not included in this report.

The number of states increase exponentially as the depth increases which account for large number of states even for small number of entries in the table. Though one bit trie is optimum for storage and update purposes but it result in large size of FSMs and thus we have to use better algorithms like *recursive flow classification* to develop Finite State Machines.

Chapter 5

Generation of FSMs using Recursive Flow Classification algorithm

The main aim of this algorithm is to perform the mapping of the packet header to *Class Identity* (*ClassID*) recursively over several stages. In the section below an overview of algorithm is presented and then in later sections the generation of FSMs using this algorithm is explained.

5.1 Recursive Flow Classification (RFC) algorithm

In the RFC[5] algorithm, the problem of finding ClassID is distributed over several stages, say P stages. Each phase consists of a set of parallel memory lookups. Each memory lookup returns a value, which is shorter than the index of memory access. Thus over P stages the output narrows down to ClassID of the packet. In our implementation memory lookup is replaced by FSMs.

In the first phase, several fields in the packet header are split into multiple chunks that are used to index multiple memories in parallel. Each of the parallel lookups yields an output value that is called the *equivalent identity* (*eqID*). In the subsequent phases, a chunk is formed by a combination of results corresponding to two (or more) chunks obtained from memory lookups in previous phase. In the final phase, we obtain the ClassID for that packet.

In our implementation, we operate in two stages, first stage is generation of tables using RFC algorithm which are converted to FSMs in second stage of implementation.Our implementation model is explained in next subsection.

5.2 Implementation Model

5.2.1 The preprocessing Stage

In the first phase (Phase = 0), F fields of the packet header are split up into multiple chunks which are used to index into multiple memories in parallel. To construct the indexing tables we have to construct a temporary *Chunk Equivalence Set* (*CES*). Each CES can be constructed in the following manner. Consider a fixed chunk of size b bits. Project the rules in the classifier onto the number line [0, 2b - 1]. Each component projects to a set of (not necessarily contiguous) intervals on the number line. The end points of all the intervals projected by these components form a set of non-overlapping intervals. Two points in the same interval belong to the same equivalence set.

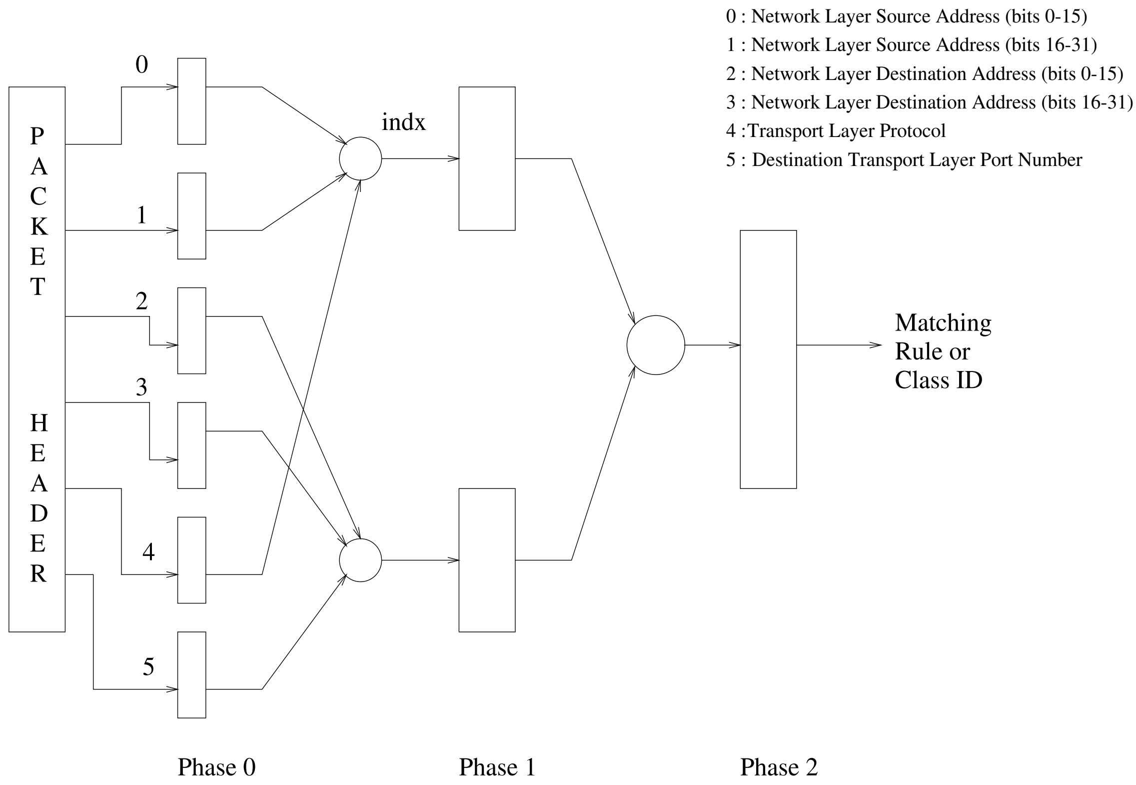


Figure 5.1: The packet flow in Recursive Flow Classification

To facilitate the calculation of the eqIDs for the subsequent RFC phases, we assign a *class bitmap* (*CBM*) for each CES indicating which rule in the classifier contains this CES for the corresponding chunk. This bitmap has one bit for each rule in the classifier. Note that the class bitmap is not physically stored in the lookup table; it is just used to facilitate the calculation of the stored eqIDs by the preprocessing algorithm.

A chunk in a subsequent phase is formed by a combination of two (or more) chunks obtained from memory lookups in previous phases, with a corresponding CES. To determine the new equivalence sets for this phase, we compute all possible intersections of the equivalence sets from the previous phases being combined.

The results of preprocessing stage are stored in tables which are later used to generate FSMs. The results are stored in form of ranges over which the eqIDs are same. In the first phase after projecting the rules on number line, the ranges over which they are eqIDs are constant are written to tables. In subsequent phases, we first calculate all the possible intersections and the eqIDs corresponding to each intersection is calculated and stored in the table.

5.2.2 Generation of FSMs from tables

The table generated in preprocessing serve as basis for generation of FSMs. We assume the packet header is available and divided into different chunks of some fixed sizes. These chunks are fed bit by bit to first phase FSMs and the second phase start only after all FSMs of first phase finishes their processing. In second phase results of previous phase are combined to generate input to this phase and so on.

Since input to different FSMs are bit wise thus we perform the mapping to eqIDs by technique called *longest prefix matching*. The range are converted to set of prefix which completely defines that range.

We convert all the tables corresponding to different phases to set of prefixes as defined in last chapter using C codes, which completely defines them. These prefix tables are later used to generate FSMs. Now the problem of finding eqIDs reduces to finding the longest matching prefix corresponding to input chunk. Since input is bit by bit serial, this finite machine is nothing but one bit trie structure. Thus codes written for generation of FSMs using one bit trie, are used to convert our prefix table to FSMs.

5.2.3 Parameters

The time taken for finding the ClassID of the packet is directly proportional to the number of phases, more number of phases implies more time requirement for packet classification. The size of FSMs are dependent on size of chunks fed to first phase. In our implementation recursive flow classification is implemented in three phases. We divide the header into six chunks, where five chunks are two bytes long and one chunk is one byte long. These chunks are fed to first phase, the six output of the first phase are divided into two groups which are combined into two chunks and fed to second phase. The two outputs of the second phase are combined and fed to final phase, which gives the ClassID.

We have assumed the packet header to containing of following fields:
Source IP address : xxxxxxxx.xxxxxxxx.xxxxxxxx.xxxxxxxx : 32 bits long
Destination IP address : xxxxxxxx.xxxxxxxx.xxxxxxxx.xxxxxxxx : 32 bits long
Source Port Number : xxxxxxxx : 8 bits long
Destination Port Number: xxxxxxxxxxxxxxxx : 16 bits long
An example of the packet header:

144.16.100.28	205.123.13.11	200	25555
Source IP Address	Destination IP Address	Source Port Number	Destination Port Number

The format of rule is as follows:
Source IP address/mask   Destination IP address /mask
range of Source Port Number   range of Destination Port Number
An example of the rule :
144.16.0.0/255.255.0.0   205.123.13.11/255.255.255.255
150-225   256-65535

The packet header is first bitwise anded with the mask and then compared with address mentioned in the rule. The use of mask allows several address to map to same address. As in above case both 144.16.100.28 and 144.16.200.30 will map to source address of above mentioned rule.

When packet comes, it is broken into six chunks. The values of chunks are as follows:
1st chunk : first sixteen bits of Source IP address in packet header
2nd chunk : last sixteen bits of Source IP address in packet header
3rd chunk : first sixteen bits of Destination IP address in packet header
4th chunk : last sixteen bits of Destination IP address in packet header
5th chunk : Source Port Number
6th chunk : Destination Port Number

The output of first phase corresponding to 1st, 2nd and 5th chunks are combined into one chunk and output corresponding to 3rd,4th and 6th chunks are combined into other chunk which are fed to second phase. This combination is chosen because these chunks are most related to each other.

5.2.4 Results

Since no database was available which can be used to check the performance of our model, we randomly generated some set of rules and ran our model on them. The result of one of our database is presented here.

We randomly generated five rules which served as our database.They are fed to C codes corresponding to preprocessing stage. The nine tables corresponding to three phases were generated. These tables were then converted to prefix table and using these prefix table FSMs were generated. The number of states in FSMs of different stages are given below:

Phase 0:

Number of states in FSM corresponding to table 0 : 34
Number of states in FSM corresponding to table 1 : 34
Number of states in FSM corresponding to table 2 : 34
Number of states in FSM corresponding to table 3 : 34
Number of states in FSM corresponding to table 4 : 18
Number of states in FSM corresponding to table 5 : 40

Phase 1:

Number of states in FSM corresponding to table 0 : 11
Number of states in FSM corresponding to table 1 : 20

Phase 2:

Number of states in FSM corresponding to table 0 : 38

The results illustrates that the number of states reduces from several hundred thousands which was in case of one bit trie to few hundreds in this case. This reduction comes at the cost of using nine FSMs instead of one. These FSMs can work in pipelined fashion to considerably increase the throughput.

One of the drawback of this algorithm is that projection of a field in rules, on a number line, can result in large number of contiguous intervals. The number of the contiguous intervals may increase exponentially with number of fields within a rule. As an example, we can see in Figure 5.2 that the set of three rules, where each rule is two dimensional, results in 36 contiguous blocks. Thus the size of FSMs in Phase 1 and Phase 2 can become very large, and whole purpose of this approach is lost.

Another drawback in this scheme is that several rules can map to single ClassID resulting in conflict. This conflict can be resolved by giving priorities to the rules. In case of conflict, the action is taken according to highest priority rule. It also suffers from the complex update of the FSM with changes in the set of rules. The changes can affect all the FSMs. In other words we may have to regenerate all the nine FSMs. Thus this algorithm will only work only when the frequency of changes in the rules is very low.

5.3 RFC algorithm for Route Lookup

The algorithm can be used for the problem of route lookup if we find some quick update method. If some entry in database changes then it will affect all the stages and in the worst case all the FSMs need to be updated. Since changes in routing database happens very frequently (especially in core routers) therefore this algorithm is not fit for route lookup problem.

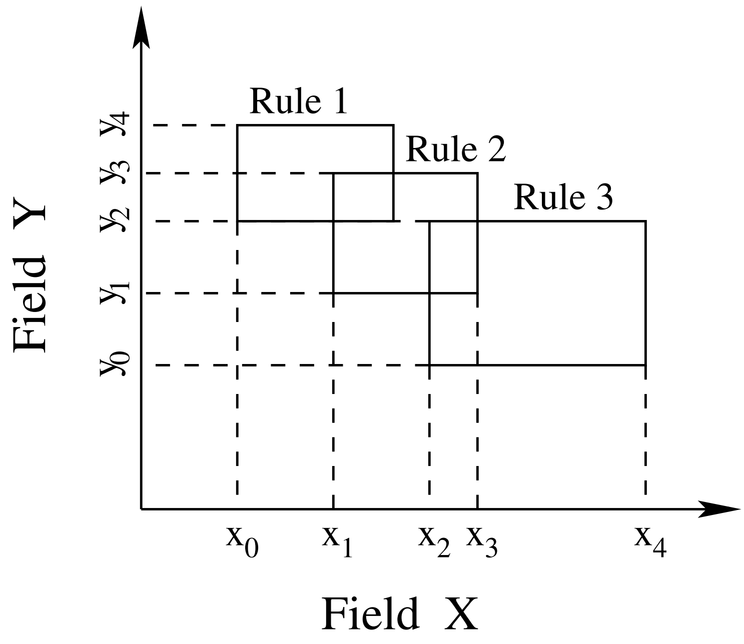


Figure 5.2: Serial Approach

5.4 Serial Approach

In the serial approach, we move from one field to another field of the rules in a serial manner. We try to utilize the information we gather by working on one field before moving on to next field. Figure 5.2 show set of three rules, where each rule is two dimensional, projected on to two orthogonal number lines. Suppose we start processing field x of the packet first, and we end up in region x_0-x_1. Next we start processing field y of the packet. Only relevant region of y for us is y_2-y_4, so we check whether field y of packet lies in that region, otherwise we have no matching rule.

The FSM generation proceed in following manner. We start with one bit trie FSM, say for field x in the above set of rules. We first convert the contiguous interval relative to field x into set of prefixes. This prefix table is used to generate FSM. At the leaf node we have either reached the dead end, *ie*, their is no matching rule or we have to start processing next field. In later case, we find out the relevant region for field y in set of rules. This region can be contiguous or non-contiguous. We convert this region to set of prefixes, generate FSM for it and attach it to the leaf node. Thus the FSM will look like FSM shown in Figure 5.3. If we have more than two field then after reaching the leaf node corresponding to field y we look for relevant region for next field and so on. One problem will be, how to indicate to bit supplier that we have to start processing the new field of packet. For this we can keep signal whose each pulse makes bit supplier to move on to the next field of the packet.

We can further optimize the size of FSM by making FSM for field y common for leaf nodes corresponding to one region. Since this FSM is one bit trie FSM therefore it has all its features like, optimum number of states and optimum number of updates if there is some changes in set of rules. But it will also suffer from large size of FSM thus we have to find some efficient way of implementing one bit trie FSM. Since the one bit trie FSM either for route lookup or packet classification is same in structure, therefore we just concentrate on one bit trie FSM for route lookup and try to find efficient way of its implementation. This is presented in later part of the

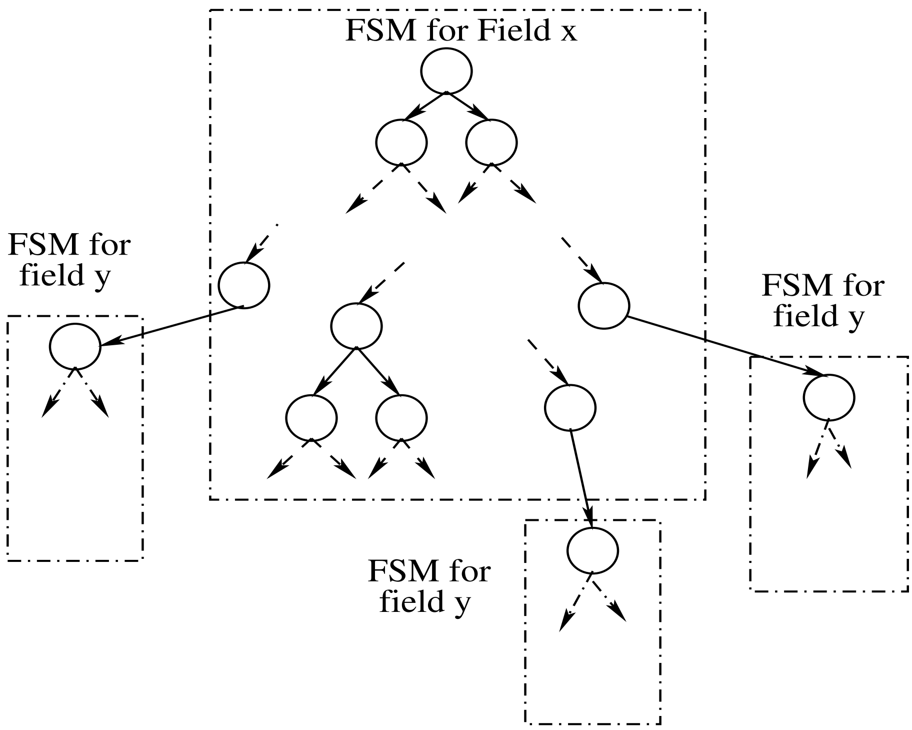


Figure 5.3: FSM according to serial approach

report. In the next chapter we explore a new technique of generating FSM for route lookup problem.

Chapter 6

FSM for Route Lookup problem

The FSM for the route lookup problem must have the following properties

- It must be linear with the number of entries in the database.
- The change in FSM due to the change of an entry in the database must be small.

One such approach is explained in this chapter.

6.1 Model

The Route Lookup problem is to find the output port number to which the packet has to forwarded depending on its destination address. The number of output port are always constant in a router. In our approach, the number of states are equal to number of output ports. Using 32 bits of the destination address, we make transition from one state to another. Being in a state implies that packet has to be forwarded to output port corresponding to that state. We start from a state *initial*. The bits are inputted bit by bit and as soon as it matches some prefix in the database of route lookup, it makes transition to the state which correspond to that prefix. The transition from that state to another state is done if some other longer prefix matches the destination address, pointing to different state. These transition continues till output port corresponding to longest prefix match is found. Then it makes transition to state *final*. The state visited before reaching state *final* correspond to the output port to which packet has to be forwarded.An example is given below to explain concept better.

Example:
Suppose the route lookup database contains following entries:
01* $\rightarrow$ Port number 1
000* $\rightarrow$ Port Number 2
011* $\rightarrow$ Port Number 3
0000* $\rightarrow$ Port Number 3

The FSM corresponding to this is shown in the Figure 6.1. It contains five states corresponding to three output ports, *initial* and *final* states. Thus number of states is linear with the number of output ports. The number of edges are proportional to number of entries in the database. There are four edges corresponding to the four entries in the database. The rest of the edges leads to state *final* after finding the longest prefix match.

The FSM so generated is linear to number of entries in the database and to the number of output ports in the router. The transition from one state to another is done by prefix matching.

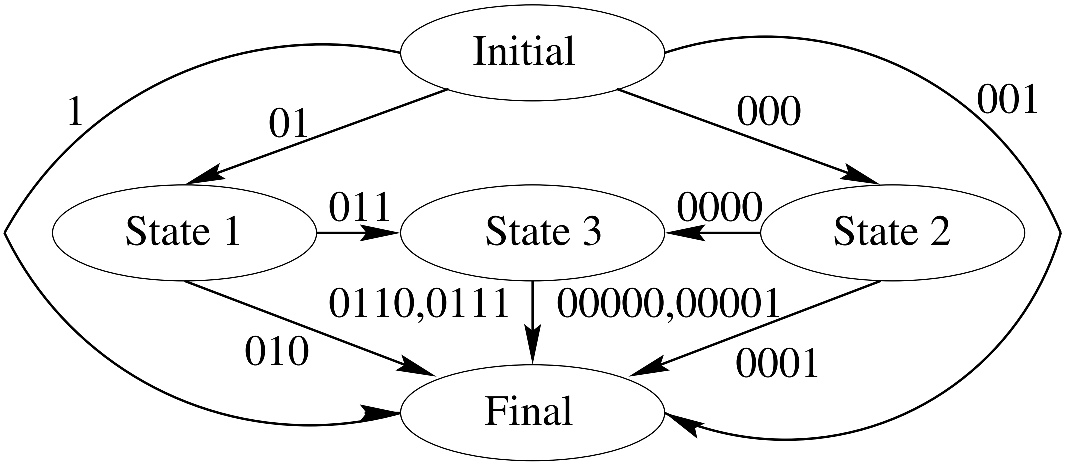


Figure 6.1: FSM for database

The prefix can be completely defined by its value and its length. As an example prefix ‘01’ can be completely defined by value equal to one and length equal to two. In our implementation these two attributes are used for prefix matching. We can also use string matching for this purpose.

The number of states is equal to number of ports and therefore it never changes. If some entries in the database changes, then few edges leaving or entering the states are changed. The transition from one state to another state is done only when the destination address matches some longer prefix, otherwise it remains in same state. If output port corresponding to prefix ‘0000’ and ‘00000’ are same, then this will result in self loop. As there is no transition corresponding to self loop thus number of edges is proportional to number of entries minus number of self loops. It is found that there are considerable amount of self loops in the database. Thus there is considerable decrease in number of edges after removing these self loops. In mae-east database the number of edges reduces from 38,623 to 35,306 and in funet database it reduces from 41,709 to 26,114.

The update of this FSM require few changes in the edges and only few states are affected. Suppose we change the output port corresponding to some prefix, then the edges corresponding to that prefix transition and next longer prefix are affected. As an example if output port corresponding to prefix ‘01’ becomes Port number 2, then edge corresponding to prefix ‘01’ comes between state *initial* and *state 2* and edge corresponding to ‘011’ comes in between *state 2* and *state 3* and edge corresponding to ‘010’ comes in between *state 2* and *final*. If one prefix is added then two edges will be added and two states will be affected until and unless it is a self loop in which case none will be affected. Similarly if some prefix is removed, then two edges corresponding to that prefix are removed and an edge between last matching prefix to next matching prefix is added. Thus two states and three edges are affected. Again if that prefix was a self loop, no change was required. Therefore while making any update we must always take care of self loops if one exist corresponding to that prefix.

All these features comes at the cost of very complex transition from one state to another. These complex transition can be implemented by implementing each state as a small FSM, *ie*, from outside they look like state while inside of each state there are smaller FSM carrying out these transitions. The breaking of these transition to smaller FSMs are explained in next subsection.

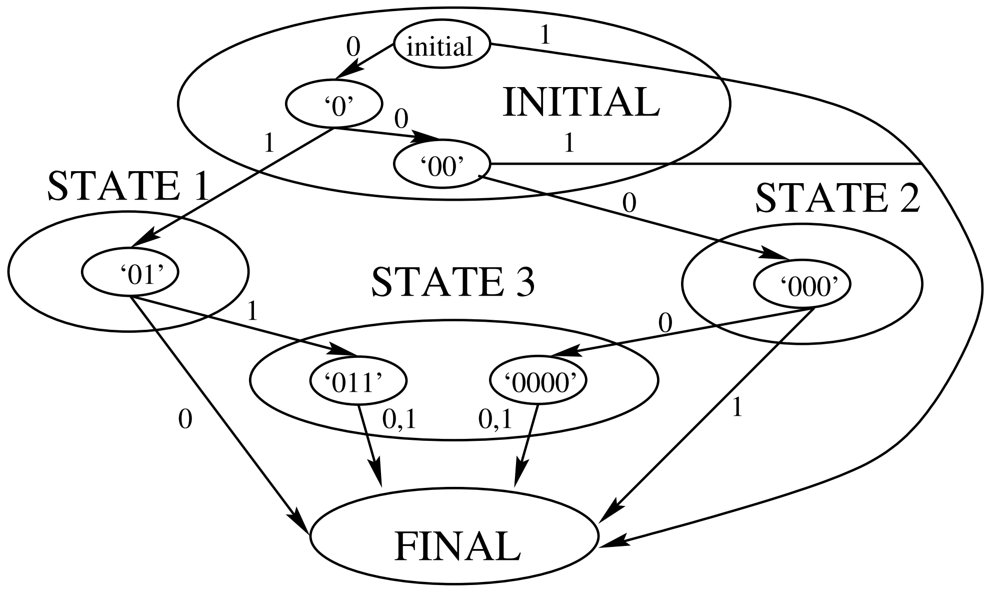


Figure 6.2: States broken to smaller FSMs

6.1.1 Breaking states into smaller FSMs

The idea will become more clear with an example. The Figure 6.2 illustrates the idea of breaking states into smaller FSMs using the same database used earlier. The transition is done using one bit at a time.

The FSM shown above contains five big states that are Initial, State 1, State 2, State 3 and Final. Each of this state are small FSM inside. These FSMs contain linked or unlinked states. The small FSMs are similar to one bit trie structure which using input bits moves from one state to another which may lie in same FSM or in different FSM. This transition continues till state Final is reached and last visited state is the port number to packet has to be forwarded.

If we closely look at the FSMs generated above, they are in-fact one bit trie FSM broken into smaller pieces using output ports as the criteria. All the states in one bit trie FSM for which the output port is same are clustered together into one FSM. The update process will be same as explained earlier but they can affect a large number of states in smaller FSMs. We used this criteria for breaking the large one bit trie FSM for different databases into smaller FSMs. The result of the implementation are presented in the next section.

6.2 Results

The method described in Section 6.1, of breaking large FSMs was carried out on two databases available. The results are as follows:

Mae-east:
number of entries in database : 38623
number of entries after removing self loops : 35306
total number of small FSMs : 61
maximum number of states in small FSMS : 106544
minimum number of states in small FSMs : 1
average number of states in small FSMs : 2649

total number of states in all FSMs : 161605

Funet:

number of entries in database : 41709
number of entries after removing self loops : 26114
total number of small FSMs : 21
maximum number of states in small FSMS : 52135
minimum number of states in small FSMs : 1
average number of states in small FSMs : 3772
total number of states in all FSMs : 79227

The results illustrates this criteria of breaking FSM into smaller FSMs gives very unevenly balanced smaller FSMs. Some FSMs contains very large number of states while most of them contains very few states. Thus this way of implementing complex transitions perform poorly. This technique will be useful if we can find some efficient method for implementing the complex transitions of the FSM.

Another approach similar to one described in Section 6.1 is to have different FSM instead of different state for each output port. We first parse the database and break it up into smaller databases. Each smaller database contain prefixes resulting in the same output port. We generate FSM for each database. Thus the number of FSMs is equal to number of output ports in the database. Each FSM will output *high* signal if some matching prefix exists in its database for the inputted bit stream else results in *low* signal. Several FSMs may result in *high* signal after processing, therefore we require a resolution mechanism. We have to find longest matching prefix, therefore the FSM which has used maximum number of inputted bits before making signal high, correspond to the port to which packet should be forwarded. As this approach is not much different from previous approach therefore the FSMs in this case can be very unevenly balanced. This approach has not been explored and is left as future work.

Since the transitions of one bit trie structure are simple and easy to implement therefore in following chapters we will concentrate on the FSM generated using one bit trie FSM. The FSMs generated for mae-east and funet databases using one bit trie algorithm are quite large. Therefore for the implementation we will have to decompose it into smaller FSMs. In the following chapter we will concentrate on some standard structures within FSM generated for the mae-east and funet databases.

Chapter 7

Characteristics of One Bit Trie FSM

There are several standard structures like completely balanced trees, paths, etc, which can be efficiently decomposed in to smaller FSMs at the cost of slightly complex transitions. There decomposition can reduce the size of FSM by considerable amount. In this chapter we will try to find the existence of such structures in one bit trie FSM.

7.1 Completely balanced tree structure

We will start with finding complete tree structure within the generated FSMs. The complete tree structure containing 2^x states can be decomposed to two FSMs each containing $2^{\frac{x}{2}}$ states. We will begin with the method used to decompose the complete tree structures. Then the results will be presented.

7.1.1 Decomposition method[9]

The objective is to decompose a completely balanced tree of size 2^x into two smaller trees each with size nearly $2^{\frac{x}{2}}$.

For partition 1:
1. include the root (level 0) and it's children (level l=1) into one block.
2. level l = 1.
3. if max level == l then stop
4. for all blocks made at previous step do
   take left child of nodes of level l; put them in one block.
   take both children of these children; put them in the same block.
   take right child of nodes of level l; put them in one block.
   take both children of these children; put them in the same block.
5. l = l +2 ; go to step 3.

For partition 2:
The method is same as partition 1 except that we put only the root and not it's children are in the first block

We will get two completely balanced trees each with $\frac{x}{2}$ levels as shown in Figure 7.1. Hence the total number of nodes in both the trees will be of the order of $2 \times 2^{\frac{x}{2}}$.

Databases	Mae-east	Funet
Total number of completely balanced tree structures	114,695	64,154
number of complete tree structure containing 511 states	1	0
number of complete tree structure containing 255 states	0	0
number of complete tree structure containing 127 states	5	5
number of complete tree structure containing 63 states	18	26
number of complete tree structure containing 31 states	115	124
number of complete tree structure containing 15 states	603	867
number of complete tree structure containing 7 states	3,331	3,939
number of complete tree structure containing 3 states	19,791	18,897
number of complete tree structure containing 1 states	90,831	40,296

Table 7.1: Completely balanced tree structures in one bit trie FSM of mae-east and funet databases

Now the two partitions are orthogonal. Any completely balanced tree can be represented as shown in the example in Figure7.1. In the representation at any node left or upper edge goes to the left sub-tree and right or down edge goes to the right sub-tree. The rows correspond to the blocks of partition 1 and columns correspond to partition 2 that would be generated by the given algorithm. Since the rows and columns intersect at only one point the partitions are orthogonal.

Since the decomposition results in completely balanced trees the same algorithm can be recursively applied to reduce the total cost in terms of states. After n decompositions the total cost will be of the order of 2^nx$2^{\frac{x}{n}}$.

7.1.2 Results

The task in hand was to find about the existence of completely balanced tree structure in one bit trie FSM, their sizes and the number of such structures. In the Table 7.1, we present the results of our simulation.

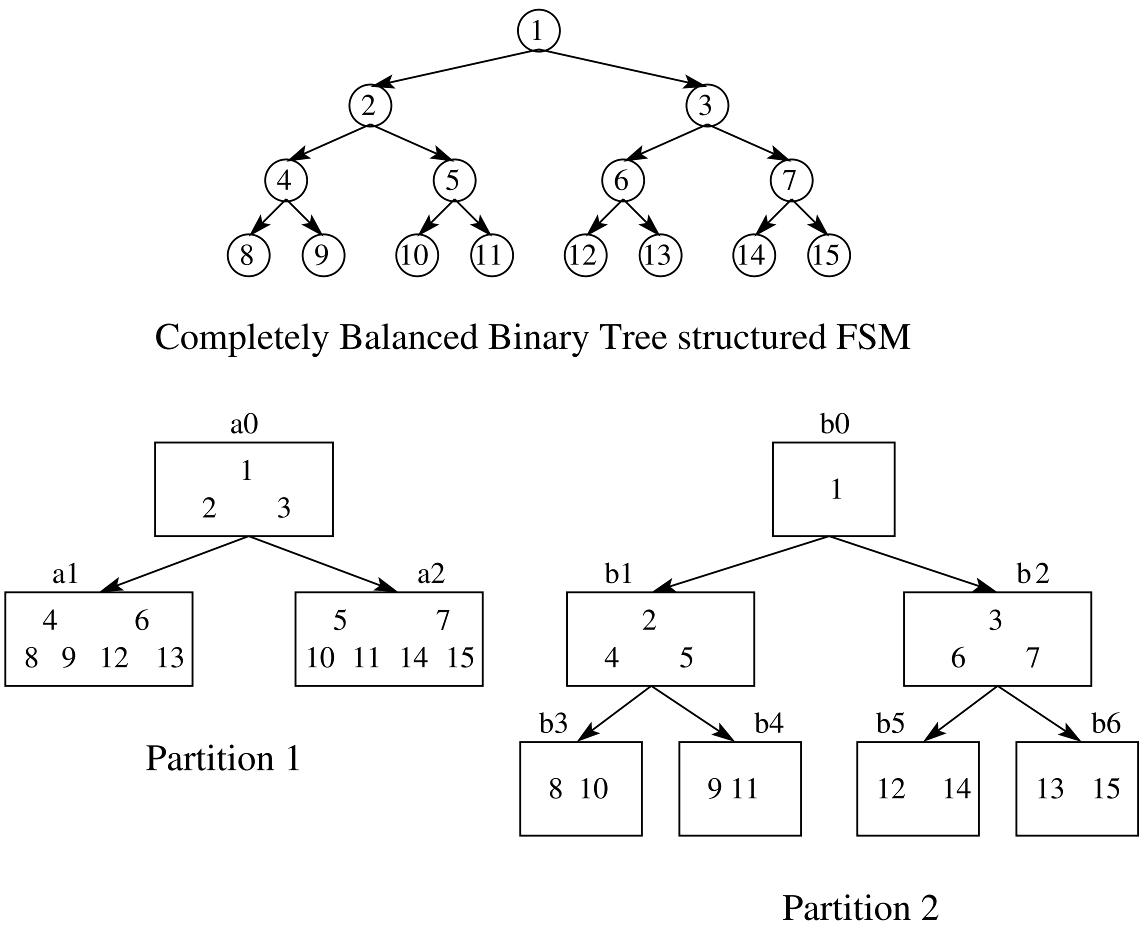
The results illustrates there are lot of paths in the one bit trie FSM as the number of complete tree structures containing only one state is very high. Since paths can also be efficiently broken in to smaller FSM therefore we will try to break our FSM into completely balanced tree structure and paths.

7.2 Paths

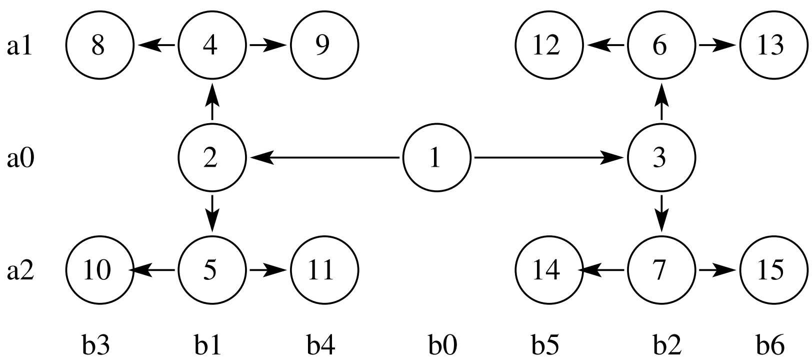
The FSM containing a path of size n^2 states can be broken into smaller FSM each containing n states. We will first present the methodology to break the path in to smaller FSMs and then present the results of breaking the FSMs into paths and completely balanced tree structures.

7.2.1 Decomposition method

The objective is to decompose path containing n^2 states into two FSM each containing n states. The method is depicted in Figure 7.2. We project the path of n^2 states onto two orthogonal axis each containing n points. The Figure 7.2 illustrates decomposition of a path containing 16 states to two partitions *Partition X* and *Partition Y* . Each partition contains $\sqrt{16}$ states.



(a) Decomposition of completely balanced binary tree structured FSM



(b) Grid representation

Figure 7.1: Decomposition of completely balanced binary tree structured FSM

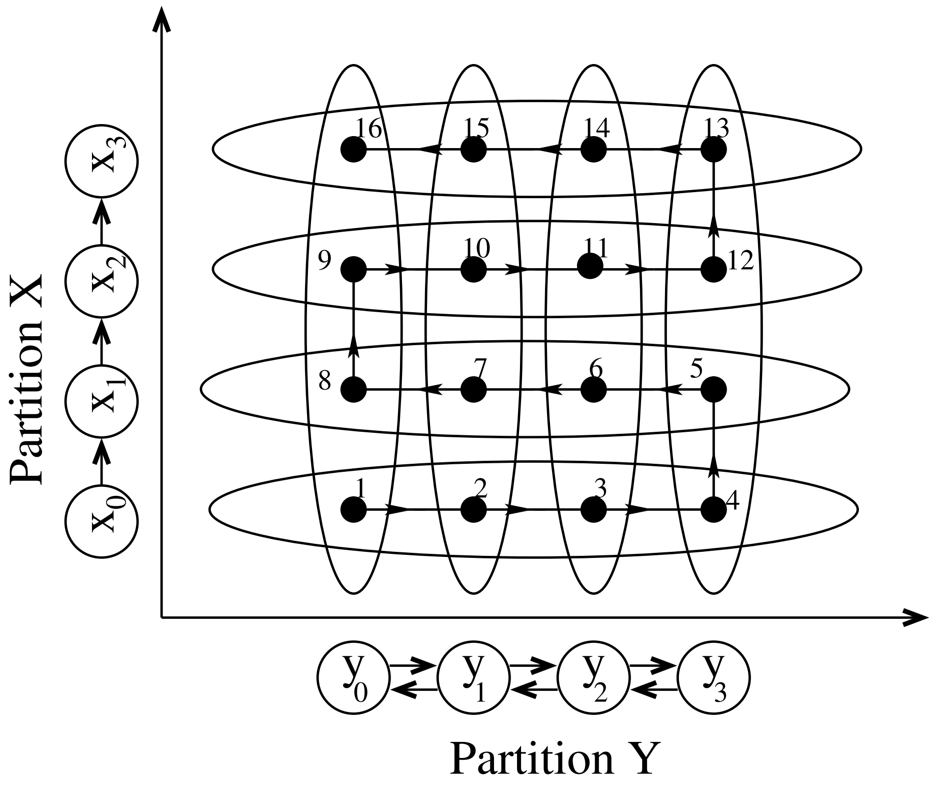


Figure 7.2: Decomposition of path

Since states in partitions intersect at unique points thus the partition is orthogonal. It can be applied recursively on the FSMs as each decomposition results in smaller paths.

7.2.2 Results

We search for the completely balanced tree structure and paths within one bit trie FSM. All the FSMs containing single states are removed because they become part of paths. The number of FSMs containing more than one state remain same. The Table 7.2, presents the simulation result of number of paths.

The results illustrates the paths contain very small number of states thus there will not be much difference between n^2 and 2n. Thus there will not be any optimization in this case.

Even one sided tree structures can be implemented efficiently, therefore we broke one bit trie FSM into completely balanced tree structures, paths and one sided tree structure. But the results were poor in this case also.

We were unable to find any characteristics in one bit trie FSM which can be exploited for efficient implementation. The FSM generated for the databases are huge and cannot be implemented as a whole therefore we broke those FSMs topographically. This is explained in detail in next chapter.

Databases	Mae-east	Funet
Total number of paths	32,767	24,185
number of paths containing 16 states	2	1
number of paths containing 15 states	2	3
number of paths containing 14 states	6	2
number of paths containing 13 states	6	5
number of paths containing 12 states	33	8
number of paths containing 11 states	121	15
number of paths containing 10 states	393	47
number of paths containing 9 states	862	129
number of paths containing 8 states	1,663	203
number of paths containing 7 states	2,042	416
number of paths containing 6 states	2,787	721
number of paths containing 5 states	3,128	1,261
number of paths containing 4 states	4,290	2,467
number of paths containing 3 states	6,253	5,430
number of paths containing 2 states	11,179	13,477

Table 7.2: Paths in one bit trie FSM of mae-east and funet databases

Chapter 8

Topographical breakdown of One Bit trie FSM

One possible solution of generating smaller FSMs is to break large FSM generated using one bit trie in to smaller FSMs according to our implementation capacity. As an example, breaking one bit trie FSM topographically into smaller FSMs is shown in Figure 8.1. The maximum number of states in this case is fixed to three. Our criteria of topographical breaking is to minimize total number of edges coming in or going out of FSMs.

8.1 Algorithm

In our algorithm we break the FSM into smaller lobes. As shown in the Figure8.1, each lobe has only one incoming edge. The number of outgoing edges equal to number of children of end nodes in the FSM. The algorithm is given below.

Step 1 : Make the *link-list* of all the deepest leaf nodes of the FSM
Step 2 : L=1
Step 3 : while *link-list* is not empty
Step 4 : node $\leftarrow$ remove and assign first element of link list
Step 5 : If node is included in [FSM_1 to FSM_{L-1}] then goto Step 3

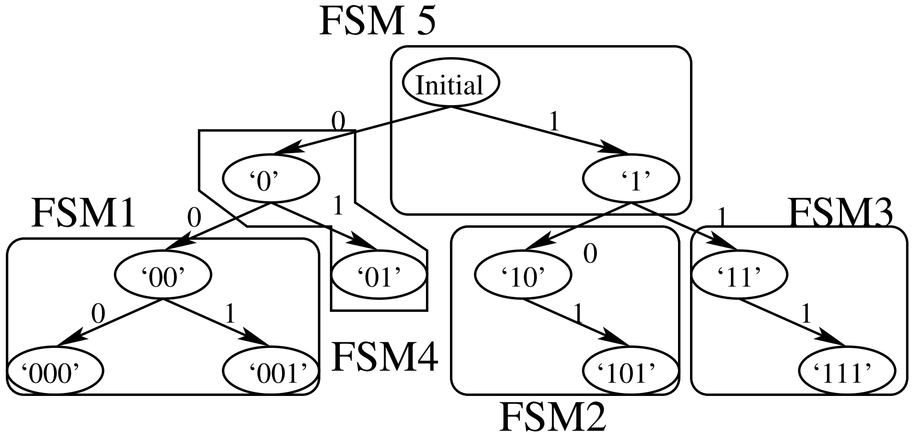


Figure 8.1: Breaking one bit trie into smaller FSMs

Step 6 : $\text{FSM}_L \leftarrow \emptyset$
Step 7 : $\text{FSM}_L \leftarrow \text{FSM}_L \bigcup$ node
Step 8 : size$\leftarrow$ 0
Step 9 : increment $\leftarrow$ number of children of parent(node) not included in FSM_L + 1
// parent(node) implies parent of the node
// 1 is added for parent(node)
Step 10 : if (size + increment < MAX_SIZE) then
Step 11 : $\text{FSM}_L \leftarrow \text{FSM}_L \bigcup$ children of parent(node) not included in $\text{FSM}_L \bigcup$ parent(node)
Step 12 : node$\leftarrow$parent(node)
Step 13 : goto Step 8
Step 14 : else
Step 15 : add one of the deepest node among children of parent(node) not included in FSM_L
Step 16 : L$\leftarrow$L+1
Step 17 : goto Step 3

Figure 8.1 illustrates the result of the algorithm when algorithm is run on the one bit trie FSM. At the beginning, link-list will contain node with prefixes '000', '001','101' and '111'.

In the one bit trie, flow of control is in downward direction, *ie*, after visiting a node the control is transferred to its child node. Thus the smaller FSMs generated by breaking large FSM are not interactive, after traveling through one FSM the control is transferred to its child FSMs and the current FSM can start processing next packet. Thus FSMs can work in pipelined fashion and will have greater throughput compared to the one bit trie FSM. The time spent in one FSM is dependent on its depth. To operate these smaller FSM efficiently in pipelined fashion all of them must have the same depth so that time spent in all FSMs are same. The highest throughput will be achieved in case when all FSMs have one bit depth.

In our implementation we have not pipelined our smaller FSMs. It will start from initial FSM and traverse path through different FSMs depending on bit pattern. The initial FSM does not start processing new destination address until the longest matching prefix is found.

In the next section, we present a greedy algorithm which is used to decompose the smaller FSMs generated by breaking one bit trie FSM topographically.

8.2 Greedy algorithm

We wish to find two orthogonal partitions *Partition X* and *Partition Y* such that the number of edges in the partition is minimum. The greedy algorithm[6] attempts to minimize the number of edges in each partition. It is obvious that, if two elements are in the same block of *Partition X*, then they must be in different blocks of *Partition Y*, otherwise their meet will not be orthogonal. The greedy algorithm builds *Partition X* by forcing tightly connected states into the same block, so that the edges between them are replaced by a self-loop. While building the second *Partition Y*, states are added one at a time to blocks by doing a local search (on assignments of the state to blocks)to determine which assignment creates the minimum.

The algorithm is as follows:

Inputs :
G(S; Es) = State Transition Graph
Q = Set of states ($\|Q\|$ = N)
Es = Set of edges

$n_1 > 0$, $n_2 > 0$, $n_1 \times n_2 \geq N$
Outputs :
Partition X, $||PartitionX||$<n_1, $||X_i||$<n_2
Partition Y, $||PartitionY||$<n_2, $||Y_i||$<n_1
Steps :
G_Y = Graph corresponding to *Partition Y*
$V_{left} \leftarrow S$
$X_i \leftarrow \emptyset, 0 \leq i < n_1$
$Y_j \leftarrow \emptyset, 0 \leq j < n_2$
$i \leftarrow 0$
while($i < n_1$)
    while($||X_i|| < n_2$)
        v $\leftarrow$Find_most_adjacent_state(X_i, V_{left}, G)
        $X_i \leftarrow X_i \bigcup$ v.
        $V_{left} \leftarrow V_{left}$ - v
        Put_in_most_suitable_block(v,G,*Partition Y*,G_Y)
    $i \leftarrow i + 1$

Find_most_adjacent_state :

In our FSM each state is linked to its parent so all the states has equal bindings. So, we try to put parent and its children in same X_i so that edge between them just reduces to self loop in Partition X.

Put_in_most_suitable_block :

In Partition Y we put parent and its children in same Y_i if they are in different X_i in Partition X or else we try to put child in that Y_i for which there is already a multi-edge between Y_j(parent) and Y_i(child).

In our implementation, we first topographically break our one bit trie FSM into smaller FSMs fixing some maximum size. Then we decompose the smaller FSMs using above mentioned greedy algorithm. These FSMs are then converted in to VHDL codes. The simulation results of the VHDL codes are presented in next chapter.

Chapter 9

Simulation results

In this chapter, we present the simulation results of One bit trie FSM implementation in VHDL. As mentioned in the previous chapter, we first topographically broke the one bit trie FSM into smaller FSMs, fixing some maximum size for them. Then, these smaller FSMs are decomposed into a machine containing two partitions using greedy algorithm, explained in previous chapter.

Each machine has following input ports, clock, reset, signal$_i$ and bit. The clock and reset are standard ports. The signal$_i$ starts the corresponding machine. The bit is port where bit corresponding to destination address are supplied. The input is bit by bit. The output ports are done$_i$, match$_i$ and signal$_j$ ($j \neq i$). The signal at port done$_i$ is set high when longest prefix match corresponding to destination address is found. The signal at port match$_i$ signifies whether we have found any prefix match in this machine or not. The signal$_j$ is used to trigger the children FSMs. The number of signal$_j$ port is equal to number of children machines controlled by this machine. The Figure9.1, gives more clear picture.

In our implementation, there is collection of such machines which are connected to each other through signals. We start from initial machine and using the bit input we traverse the path through different FSMs. The done$_i$ corresponding to machine where lookup ends becomes high. Then the all machines are reseted and the processing starts on new address. At present the output port corresponding to longest matching prefix is not out-putted, only match signal is set high.

The simulation was done on *XILINX Foundation Series*. The version used were *F2.1i* and *F3.1i*. The FPGA implementation was done either on device *V1000FG680*, belonging to family *VERTEX*, or on device *V812EFG900*, belonging to family *VERTEXE*. The Table 9.1, presents

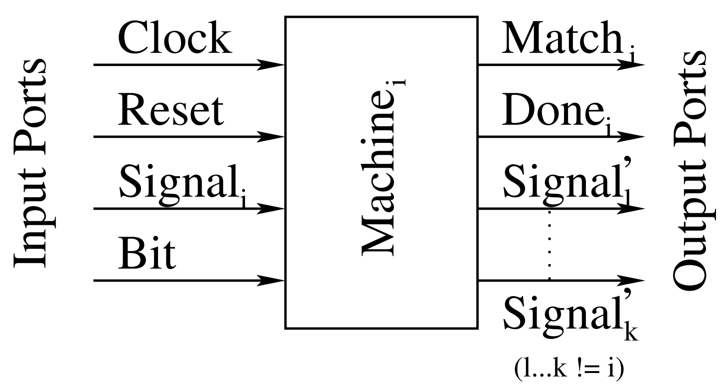


Figure 9.1: Machine with input and output ports

One bit trie FSM containing	Smaller FSMs containing max.	Flip Flops	LUTs	Slices	Number of routes	Min. Clock Period (ns)	Max. Net Delay (ns)
1851 states	1024 states	212	2,832	1,459	10,328	31.050	16.159
4125 states	1024 states	567	6,409	3,298	23,393	24.512	13.925
	2048 states	593	5,999	3,094	22,053	33.631	15.003
6082 states	1024 states	782	12,535	6,468	45,392	45.840	8.683
	2048 states	804	14,033	7,139	50,076	56.342	8.837
	4096 states	603	15,884	8,090	56,317	62.058	13.084
8365 states	1024 states	922	12,063	6,262	44,149	49.258	15.180
	2048 states	791	10,909	5,597	39,704	31.791	8.3420
	4096 states	674	9,852	4,987	35,564	39.615	12.651
16090 states	1024 states	1,916	23,580	12,288	87,014	43.240	12.002
	2048 states	1,484	20,658	10,655	74,928	41.128	16.199
	4096 states	944	17,821	9,145	64,831	48.536	11.138

Table 9.1: Simulation Results [set 1]

One bit trie FSM containing	Smaller FSMs containing max.	Flip Flops	LUTs	Slices	Number of routes
Mae-east	1024 states	20,114	245,160	130,780	925,690
	2048 states	12,877	213,320	110,100	770,000
	4096 states	6,318	179,510	93,509	658,640
Funet	1024 states	14,634	178,580	95,117	673,240
	2048 states	9,442	155,340	80,165	560,800
	4096 states	4,701	130,860	68,126	479,980

Table 9.2: Predicted Results [set 1]

one set of results.

The results illustrates that number of flip flops, LUTs, number of slices and number of routes vary linearly with number of states implemented. The plot in MATLAB confirms the linearity in results. The proportionality constant are different in all the cases. It decreases as the number of states in smaller FSM increases. The Minimum clock Period and Maximum net delay does not vary in some particular fashion with number of states.

The devices get exhausted when we implemented FSM containing 16,090 states, thus the one bit trie FSM for Mae-east and Funet databases cannot be implemented. Since the results for flip flops, LUTs, number of slices and number of routes are linear, therefore using linear relationship we predict the results for the databases. The predicted results are presented in Table 9.2.

We cannot predict the Minimum clock period or Maximum net delay for the databases as we were not able to find any relationship. We just predict the time for route lookup using minimum clock period for FSM containing 16,090 states. We assume that the large FSM corresponding to databases are implemented in block of FSMs each containing approximately 16,000 states, such that, each block can be accommodated on one device. There will be some latency introduced due to interaction of the devices. The minimum clock period in case of FSM containing 16,000 states is approximately 50ns. In the worst case, the lookup will take 32 steps for *IPv4*, *ie*, 1,600ns. Assume other latencies to be the order of 400ns, thus in the worst case it will take 2,000ns for route lookup. This implies we can do 500,000 lookups per second. In the worst case, the throughput will be 128 Mbps, assuming all packets are of size 32 bytes. Though the

One bit trie FSM containing	Smaller FSMs containing max.	Flip Flops	LUTs	Slices	Number of routes	Min. Clock Period (ns)	Max. Net Delay (ns)
1851 states	1024 states	170	2,821	1,476	10,379	24.865	9.309
4125 states	1024 states	419	6,094	3,226	22,552	26.141	9.923
	2048 states	345	5,638	2,975	20,855	27.985	7.396
6082 states	1024 states	534	8,702	4,536	32,035	27.727	8.668
	2048 states	456	8,059	4,194	29,598	28.606	19.486
	4096 states	315	6,883	3,576	25,328	35.701	9.476
8365 states	1024 states	752	11,611	6,106	42,920	27.370	7.458
	2048 states	575	10,641	5,565	39,056	66.895	21.723
	4096 states	420	9,390	4,877	34,386	35.995	8.972
16090 states	1024 states	1,502	23,094	12,182	85,204	42.169	14.033
	2048 states	1,062	20,409	10,749	74,755	36.190	13.029
	4096 states	722	17,413	9,070			

Table 9.3: Simulation Results [set 2]

One bit trie FSM containing	Smaller FSMs containing max.	Flip Flops	LUTs	Slices	Number of routes
Mae-east	1024 states	16,013	245,500	129,590	905,940
	2048 states	10,493	213,960	112,900	802,370
	4096 states	7,035	181,800	94,872	686,900
Funet	1024 states	11,644	178,540	94,241	658,850
	2048 states	7,654	155,710	82,154	583,370
	4096 states	5,136	132,340	69,059	499,800

Table 9.4: Predicted Results [set 2]

worst case performance seems to be very poor but it is a very crude estimate. The performance will certainly improve when we move to the ASIC implementation.

In the above case the two partitions in the machine are fixed to have states equal to square root of the maximum number of states in smaller FSMs. In next set of simulation, we changed the number of states dynamically to square root of number of states in smaller FSMs. The results are presented in Table 9.3.

Even in this case the results are linear and proportionality constants are smaller compared to previous case. There is also improvement in Minimum clock period and Maximum net delay. Thus, this strategy is surely an improvement in performance. The predicted results for databases are presented in Table 9.6.

There can be the case that lot of resources are wasted in routing signals from one machine to another. To check it out we ran simulation for FSM containing approximately 1,024, 2,048 and 4,096 states. They were implemented as one piece, *ie*, they were not topographically broken to smaller pieces. The results are presented in Table 9.5.

The results illustrate that most of resources are utilized within the machine and not in routing signals between the machines. The results also illustrate that minimum clock period increases tremendously as number of states increases to 4,096 states. Thus it is useful to keep the size of smaller FSMs bounded by 2,048 states.

We can gather from above sets of results that maximum net delay is a considerable proportion of minimum clock period. Thus the transmission of signals between machines does put constraint on clock period. Thus we require a programmable unit which allows interaction

One bit trie FSM containing	Flip Flops	LUTs	Slices	Number of routes	Min. Clock Period (ns)	Max. Net Delay (ns)
1023 states	70	1,367	703	4,968	25.701	9.472
2047 states	98	2,405	1,238	8,743	27.903	9.311
4092 states	134	4,269	2,196	15,590	51.482	13.688

Table 9.5: Simulation Results [set 3]

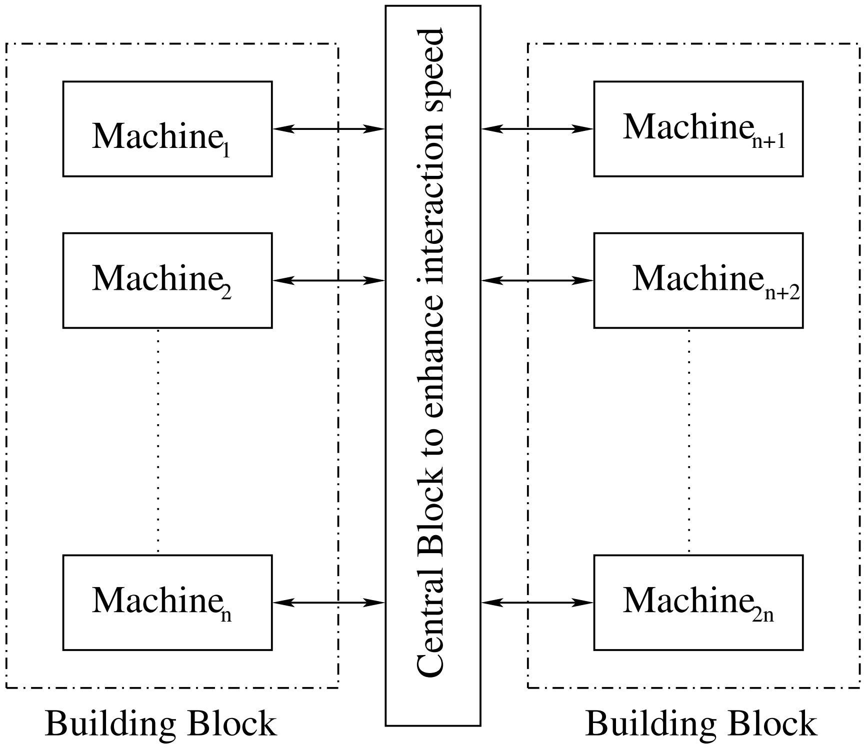


Figure 9.2: The architecture of route lookup machine

between machine at very high speed. The architecture will then look like architecture shown in Figure 9.2.

The performance will enhance considerably when the implementation is done on ASIC. In next section, we discuss the performance on ASIC implementation.

9.1 ASIC implementation

The architecture for the ASIC implementation of string matching FSM has been reported in [8]. Using this model, we can project the size of the ASIC for our FSM block. The size was dependent on the number of multi-edges in the two partitions and the number of states in those partitions. The formula [8] is given below.

Length = (multi-edges in partition x + multi edges in partition y) $\times$ 108λ
width = 4 $\times$ number of states in partition x (= partition y) $\times$ 76λ
108λ = length of one cell.
76λ = width on one cell.

Database	Smaller FSMs containing max.	Max. number of Partition x	multi-edges in Partition Y	Number of FSMs	Length (cm)	Width (cm)	Area (cm^2)
Mae-east	1024 states	59	226	260	0.385	0.122	12.164
	2048 states	82	386	125	0.632	0.175	13.805
	4096 states	122	632	65	1.018	0.243	16.091
Funet	1024 states	58	228	187	0.386	0.122	8.780
	2048 states	79	398	88	0.644	0.175	9.905
	4096 states	112	666	46	1.050	0.243	11.750

Table 9.6: Area for ASIC implementation of mae-east and funet databases.

λ= .125$\times$ 10^{-6} for 0.25μ technology.
Number of states in partition x = Number of state in partition y =$\sqrt{number\ of\ states\ in\ block}$

In the Table 9.6, we present the area predicted for the mae-east and funet databases. The values does not change much in case, where, the number of states in partition x and partition y are dynamically allocated.

The operational frequency can also be predicted for the ASIC implementation but that work is not done yet. In case of string matching[8], the ASIC implementation worked ten times faster than the FPGA implementation. If our case has improvement of same amount then the worst case throughput will be 1.28Gbps. Thus, in average case our architecture can operate at approximately OC48c line rate (2.5Gbps).

In the ASIC implementation, one dimension is linearly dependent on the number of multi-edges and other dimension is linearly dependent on number of states in partition x/partition y. Thus if we find a way to decompose the FSM resulting in either smaller of multi-edges or reduction in number of states in partitions then area of our ASIC implementation will decrease proportionally. It may also have positive effect on the operational frequency.

Even though FPGA implementation results are not very good, the prediction for ASIC implementation seems to be very promising. There is a possibility of further enhancement in performance by improvising the ASIC implementation and making it more specific to our problem. The performance can be improved by several times by introducing pipelining in our architecture. This is possible as the flow in one bit trie FSM is in downward direction and control is never returned to any state which is already traversed. Thus, when machine completes its processing on one destination address and transfer control to another machine, it can immediately start processing another address. This improvement will come at the cost of complicated control of all the smaller machines. In next chapter, we will conclude the report with Conclusion and Future Work.

Chapter 10

Conclusion and Future Work

The purpose of our work was to come out with an innovative design for packet forwarding engine. We learned that major bottleneck in its performance are route lookup and packet classification. Several solutions exists for these problems, both in hardware and software. The motivation was to combine the best features of these software and hardware solutions. We proposed an architecture containing a *Hardware component* and *a Processor*. The processor used the existing software solutions to generate FSMs for above mentioned problems. These FSMs are implemented on hardware component.

We started with using one bit trie algorithm to generate FSM for route lookup and packet classification. The size of FSM generated using this algorithm were very large and cannot be implemented efficiently. Thus we tried Recursive Flow Classification algorithm to generate a FSM. This algorithm reduced the size of FSM for packet classification by considerable amount but at the cost of increase in number of FSMs. The algorithm was not useful in case of route lookup as it perform poorly in case of updates. Then we presented a new scheme for generating FSM for route lookup problem but it suffered from the problem of complex transition.

Then we tried decomposing one bit trie FSM into smaller FSMs but we were unable to find any characteristics in one bit trie FSM which can be exploited for efficient implementation. Thus we topographically broke the large one bit trie FSM into smaller FSMs and then decomposed these smaller FSMs using greedy algorithm.These machines were then implemented in VHDL.

The VHDL codes were simulated on *XILINX foundation series*. The FPGA implementation can only be done for FSM up-to size of 16,000 states due to size constraint of devices. The results were linear to the number of states in the FSM except for minimum clock period and maximum net delay, which varied randomly . Using the linear relationship, the results were predicted for mae-east and funet databases. We then projected results for the ASIC implementation. They seemed to very promising. In the ASIC implementation, our architecture can operate at OC48c line rates in average case.

The future work includes the timing analysis for ASIC implementation and achieving the results projected for the ASIC implementation. Several possible improvisations are presented in last chapter which included pipelining the ASIC architecture and making it more specific to our requirement. The major efforts are required in decomposing the one bit trie FSM as any improvement either in number of multi-edges or in number of states will improve our architecture in direct proportion. All these improvisations in our architecture are also left as future work. After we have find some efficient way of implementing FSM for route lookup problem then we can try implementing one bit trie FSM for packet classification according to method explained in Section 5.4.

Bibliography

[1] V. Srinivasan and G. Varghese, "Fast Address Lookups using Controlled Prefix Expansion", *Proceedings of ACM sigmetrics*, June 1998.

[2] Pankaj Gupta, Steven Lin and Nick Mckeown, "Routing Lookups in Hardware at Memory Access Speeds", *IEEE infocom*, Vol 2, pp. 808-815, April 1998.

[3] Andrey Belenkiy and USA Necdet Uzun, "Deterministic IP Table at Wire Speed", *Proceedings of INET*, 1999.

[4] V. Srinivasan, S. Suri and G. Varghese, "Packet Classification using Tuple Space Search", *Proceedings of ACM SIGCOMM*, pp. 135-146, 1999.

[5] Pankaj Gupta and Nick Mckeown, "Packet Classification on Multiple Fields", *Proceedings of ACM SIGCOMM*, pp. 147-160, 1999.

[6] Rupesh S. Shelar, Madhav P. Desai and H. Narayanan, " Decomposition of Finite State Machines for Area, Delay Minimization", *Proceedings of IEEE International Conference on Computer Design*, pp. 620-625, 1999.

[7] Douglas L. Perry, "VHDL", *McGRAW-HILL International Edition, second edition*, 1995.

[8] Prakash Ch. Vanamala, "Implementation of large FSMs by Decomposition", *M.Tech. thesis, IIT-Bombay,* January 2001.

[9] P. Shesh Murthy, Personal Communication.