IP ROUTING LOOKUP:

HARDWARE AND SOFTWARE APPROACH

A Thesis

by

RAVIKUMAR V. CHAKARAVARTHY

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2004

Major Subject: Computer Science

IP ROUTING LOOKUP:

HARDWARE AND SOFTWARE APPROACH

A Thesis

by

RAVIKUMAR V. CHAKARAVARTHY

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Jyh-Charn Liu (Co-Chair of Committee)	Rabi N. Mahapatra (Co-Chair of Committee)
A.L. Narasimha Reddy (Member)	Valerie Taylor (Head of Department)

May 2004

Major Subject: Computer Science

ABSTRACT

IP Routing Lookup:

Hardware and Software Approach. (May 2004)

Ravikumar V. Chakaravarthy, B.E., Mysore University

Co-Chairs of Advisory Committee: Dr. Jyh-Charn Liu
Dr. Rabi N. Mahapatra

The work presented in this thesis is motivated by the dual goal of developing a scalable and efficient approach for IP lookup using both hardware and software approach. The work involved designing algorithms and techniques to increase the capacity and flexibility of the Internet. The Internet is comprised of routers that forward the Internet packets to the destination address and the physical links that transfer data from one router to another. The optical technologies have improved significantly over the years and hence the data link capacities have increased. However, the packet forwarding rates at the router have failed to keep up with the link capacities.

Every router performs a packet-forwarding decision on the incoming packet to determine the packet's next-hop router. This is achieved by looking up the destination address of the incoming packet in the forwarding table. Besides increased inter-packet arrival rates, the increasing routing table sizes and complexity of forwarding algorithms have made routers a bottleneck in the packet transmission across the Internet.

A number of solutions have been proposed that have addressed this problem. The solutions have been categorized into hardware and software solutions. Various lookup algorithms have been proposed to tackle this problem using software approaches. These approaches have proved more scalable and practicable. However, they don't seem to be able to catch up with the link rates. The first part of my thesis discusses one such software solution for routing lookup.

The hardware approaches today have been able to match up with the link speeds. However, these solutions are unable to keep up with the increasing number of routing table entries and the power consumed. The second part of my thesis describes a hardware-based solution that provides a bound on the power consumption and reduces the number of entries required to be stored in the routing table.

To my Parents Raghavan and Rajalakshmi
and brother Ravikiran

ACKNOWLEDGEMENTS

I would like to thank Dr. Liu and Dr. Mahapatra for their invaluable guidance; moral support and constant feedback that helped me complete this thesis work. Their experience and background in this field has been the prime motivation for me to pursue this research topic. Dr. Liu motivated me to aim very high and compete with the best in the field. Dr. Mahapatra guided me throughout my thesis work and gave valuable input and feedback to help me complete my thesis.

I would like to thank Dr. Reddy for being a part of my committee and his invaluable concepts in the Networking course (ELEN 601) that not only helped me pursue my thesis but also enhanced my technical skills in the field. I am also grateful to him giving his valuable time to advise me on my thesis.

I am grateful to Sai for not only helping me with the thesis work, but also helping me in all aspects of my graduate studies. Subramonia Sarma and B.S. Kumar have shown me the true colors of friendship and offered me more than what I could possibly fathom. I would like to thank Dr. Dmitriv and Siddarth Choudary for helping me with various aspects of my thesis work. I would also like to thank Di Wu, Sei-Young, Vishi and Saurabh for helping me during some difficult times.

My parents and my brother have been a true motivation for me all through my life and will continue to remain deep in my heart forever.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

Due to the technology advancement, the packet transmission rate is ever growing. With increasing number of hosts and sessions in the Internet, the processing of packets at the routers has become a major concern. The higher link speed demands the reduction of packet processing overhead at routers. Such packet processing involves switching of data from the input to the output port and packet forwarding. While the current technology supports fast switching, packet forwarding is still a bottleneck. This is either due to a high processing cost or large memory requirement [1].

Internet addressing architecture changed from class based to class-less addressing because of depletion of IP address and exponential growth of routing tables. Initially for class based addressing, where the prefixes were of fixed length, algorithms like hashing and binary searches [1], proved to be efficient. However, with the evolution of classless addresses, the prefix lengths have been of varying length and the traditional algorithms are no longer efficient. The routing engine can be implemented in either hardware or software depending on the requirements. Using dedicated hardware each routing lookup can be done in one memory access [1]. The dedicated hardware techniques are limited by the memory requirements to store data and hence are expensive. These techniques also consume a lot of power.

The routing tables implemented in most routers today generally follow the software-based approach, as they are more flexible and adaptive to any changes in the protocol. The hardware solutions are also not scalable and hence with the emergence of the IPv6, these approaches are nearly impractical. The software approaches expect to gain speed as the processing rate is doubled every 18 months (Moore’s law).

The entries in the forwarding tables have always been increasing as the number of hosts in the Internet is increasing. From Figure 1 we see that between early 90’s to late 90’s, the number of entries in the lookup table has changed from linear to super linear. In 2002 the backbone router contain approximately 100,000 prefixes and are constantly growing [1]. A lookup engine deployed in the year 2002 should be able to support approximately 400,000-512,000 prefixes in order to be useful for atleast another 3 years. Thus, the lookup algorithms must be able to accommodate future growth.

This thesis follows the style of *IEEE Journal on Selected Areas in Communications.*

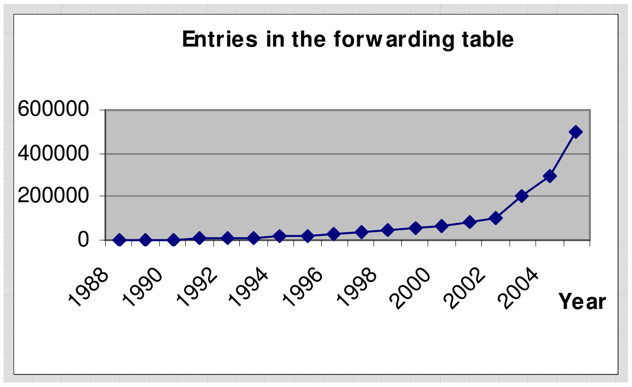


Figure 1: Year vs Entries of the Forwarding Table [9]

Apart from the lookup entries the speed of the links are doubling every year. From Table 1 it can be inferred that the links running on OC768c (approximately 40Gbps) require the router to process 125 million packets per second (Mpps) (assuming minimum sized 40 byte TCP/IP packets) [1]. For applications that do not require quality of service, a lookup or classification algorithm that performs well in the average case is desirable. This is so because the average lookup performance can be much higher than the worst-case performance. For such applications the algorithm needs to process packets at the rate of 14.1 Mpps for OC768clinks, assuming an average Internet packet size of approximately 354 bytes [1].

Table 1: Lookup Rate Required [1]

Year	Line	Line-rate (Gbps)	40B packets (Mpps)
1998-99	OC12c	0.622	1.94
1999-00	OC48c	2.5	7.81
2000-01	OC192c	10.0	31.25
2002-03	OC768c	40.0	125

Internet consists of many interconnected routers and end-hosts connected to these routers. The time taken to transfer the packets between end hosts today is greatly decided by the processing of the packets at the intermediate routers. The link speeds are also constantly increasing and require the packets to be transferred at wire speeds for maximum utilization. However, today's packet processing rate is unable to keep up with the link speeds and this necessitates the need for an efficient and fast routing mechanism. Also, the number of hosts in the Internet is doubling every 3 months, which requires a scalable solution to accommodate the ever-increasing Internet traffic. The packet processing functionality of the router primarily consists of switching, forwarding, and packet classification. We focus on the forwarding functionality of the router in this paper.

The routing table is implemented using both the hardware and software approaches. The software approaches are cheaper, flexible and scalable. However the drawback of these approaches is that they cannot keep up with today's link speeds (OC768c links require router to process 125 million packets per second), while a maximum of 20 million-packet lookup per second is achieved by today's software approach as in [2]. Today's hardware approaches on the contrary have achieved lookup speed of 100 Million packets per second [1]. The main drawbacks of these approaches have been power consumption, heat dissipation, manufacturing costs and scalability.

A. Background

Following is an analysis of the different approaches with respect to lookup time (or access time) and memory utilization. We have classified the existing approaches into Trie and Non-Trie based approaches.

1. Non-Trie Based Approaches

Linear search is simplest data-structure with the linked list of all prefixes in the forwarding table. The algorithm traverses through each and every prefix and finds the longest prefix matching. Storage and time complexity is O (N). With the emergence of IPv6 this approach is almost impractical to continue. Binary search algorithms include the classical methods like the hashing and tree based approaches. These techniques are based on matching a fixed prefix length. Since one of the main issues in lookup is to find the longest prefix match, these algorithms cannot distinguish matches based on prefix length. One such algorithm is the modified binary search technique described in [3] that uses $\log_2$ (2N), where N is the number of routing table entries. These algorithms are also computation intensive and require a large storage especially as the lookup table grows. These algorithms also consume a lot of power, as they are computation intensive. Caching is a better technique than other memory reference techniques, as it is faster [4]

[5]. A fully associative memory, or content-addressable memory (CAM) [1] [6], can be used to perform an exact match search operation in hardware in a single clock cycle. Using large associative caches can increase the hit ratios significantly. But these associative memories are implemented using CAM which is a costly mechanism and not feasible. Thus the storage requirements for these become a limiting factor. Also CAM based approaches are for fixed length prefixes. A better solution is to use a ternary-CAM (TCAM), a more flexible type of CAM that enables comparisons of the input key with variable length elements.

Table 2: Binary Strings Stored in a Trie Structure

Nbr	String	Nbr	String
1	000	11	0110
2	00101	12	0111
3	010	13	10100
4	011	14	10101
5	100	15	10110
6	101	16	10111
7	110	17	11101000
8	1110100	18	11101001
9	0000	19	101000
10	0001	20	101010

2. Trie Based Approaches

Trie [7] is a general-purpose data structure for storing strings. Each string (prefix) in the routing table is represented by a leaf node in the trie. A trie is a binary tree that has labeled branches, and that is traversed during a search operation using individual bits of the search key. The left branch of a node is labeled 0 and the right-branch is labeled 1. The longest prefix search operation on a given destination address starts from the root node of the trie [1]. The remaining bits of the address determine the path of traversal in a similar manner. Figure 2 illustrates how binary strings (as represented in Table 2) are stored in a trie structure. Patricia trie is a modification of a regular trie. The difference is that it has no one-degree nodes. Each branch is compressed to a single node in a Patricia tree. Thus, the traversal algorithm may not necessarily inspect all bits of the address consecutively, skipping bits that formed part of the label of some previous trie branch.

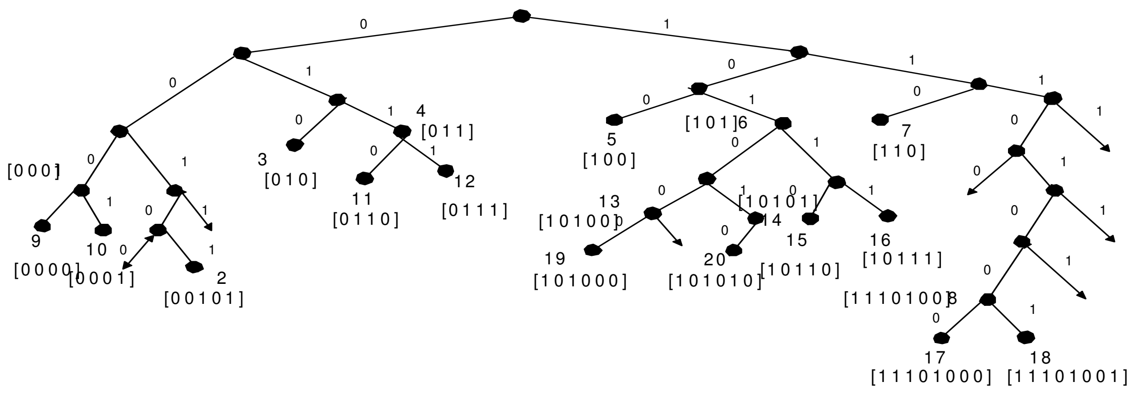


Figure 2: Radix Trie Approach

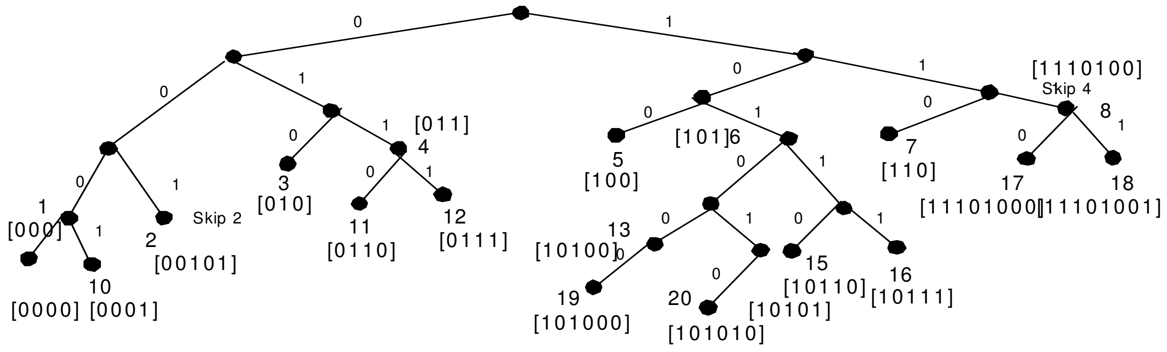


Figure 3: Patricia Trie

Patricia tree loses information while compressing chains; the bit-string represented by the other branches of the uncompressed chain is lost. Another traditional technique to overcome this problem is the *path compression* technique. A Path-compressed trie node stores the complete bit-string that the node would represent in the uncompressed basic trie (Figure 3). This compression is recorded by maintaining a skip value at each node that indicates how many bits have been skipped in the path [8]. This representation has 2n-1 nodes in the tree, where n is the number of leaf nodes in the trie [1]. The statistical property of this trie (Patricia trie) indicates that it gives an asymptotic reduction of the average depth for a large class distribution [9]. However when the trie is densely distributed this approach fails in terms of storage and processing for lookup. Level compressed trie (LC-trie) [10] is a modification to this scheme for densely populated tries. An *LC-trie* is created from a binary trie as follows. First, path compression is applied to the binary trie. Second, every node that is rooted at a complete sub-trie of maximum depth is expanded to create a 2^k-degree node [1]. This expansion is done recursively on each subtrie of the trie. It replaces the *i* highest complete levels of the binary trie with a single node of degree 2^i. This gives an expected average depth of O (log*n) for an independent random sample, where log*n is the iterated logarithm function, log*n=1+log*(logn), if n>1, and log*n=0 otherwise. All the internal nodes represented in this trie contain no information but pointers to the first child. Hence a separate vector is needed to store all possible prefix in a *prefix vector* in case of a failure in search. Also since each node is traversed again by backtracking in case of failure this is an inefficient method both in terms of processing time and storage. Thus all the variants of the trie-based approach are not storage efficient and require a lot of processing.

Several hardware approaches like [10, 11] have been proposed that use dedicated hardware. However, more popular techniques use the commercially available Content addressable memory (CAM). The CAM storage architectures have gained popularities since their search time is bounded by a single memory access. Binary CAMs allow only fixed length comparisons and hence are not suitable for longest prefix matching. The TCAM (where each bit stores a 0,1 or don't care) solves the longest prefix problem and by far is the fastest hardware device for CIDR (Classless Inter Domain Routing) routing.

The search time in a TCAM with N prefixes is O(1). The longest prefix lookup requires TCAM entries to be stored in a sorted manner. The TCAM prefix match with the lowest physical address is the longest prefix match for the given destination IP address. Such designs allow an update time of O(N), which most TCAM vendors live with. This is a costly approach especially when N is large and the updates are frequent. The update time could be easily reduced to O (N/2) by having the free pool at the middle of the array rather than at one end of the TCAM. Another attempt to include a sorting technique that uses a “Max” function on the matching entries to avoid storing entries in the sorted manner is discussed in [12]. Their approach proved expensive due to the large number of matches. Another approach includes a circuit-level optimization for faster updates at the cost of slower search time [11]. The approaches proposed in [6] partition the routing table based on prefix length and requires an update time of $O(P_n)$ memory shifts, where P is the size of the partition for the prefix length n. In [12], authors discuss two different algorithms to optimize updates in O(L/2) memory references, where L=32 (IPv4), and O(D/2) prefix shifts respectively, where D is the maximum length of the chain in the trie. This is by far the best approach for incremental updates. This however requires a TCAM manager or an auxiliary trie data structure that consumes memory.

Since the routing entries today are increasing super-linearly (routing tables support approximately 125000 entries and are estimated to contain about 500000 entries by 2005), the need for optimal storage is an important concern. The CAM vendors are recently advertising to handle 8000-128000 worst-case numbers of prefixes, considering allocators and deallocators into account [8]. Hence, attempts to optimize the storage based on prefix properties have been recently proposed by the many authors including [4, 7]. These approaches however have a high computational overhead and are not scalable. Another drawback of the existing approaches is that every lookup involves searching the entire CAM for the match. This results in a huge power consumption and heat dissipation.

II. MODIFIED LC-TRIE

A. Research Objectives

The main motivation behind our approach is as follows

1. Reducing the lookup time: The software approaches for lookups can be categorized into the methods based on tries and hashing. Many of the trie-based approaches like [10] [11] take 6-7 memory references for lookup, while hash technique combined with Tries in [3] takes 5 memory references. Our goal is to modify the trie-based approach to reduce the number of memory references for each lookup.
2. Reduce routing table storage: Implementing the routing table using trie [7] could be an expensive process. This storage is not efficient especially when the number of nodes is large and the depth is long. In this work, our aim is to reduce the storage requirements by eliminating unnecessary and redundant data.
3. Ability to handle large routing tables: Not all the algorithms suggested are scalable and handle real life routing tables. Most of them support routing table sizes for the current needs. However, an algorithm needs to be scalable so that it supports the routing table storage requirements for atleast the next 3 years.

In this section, we describe a scalable time efficient level compression technique. The approach presented here is motivated to minimize the access time and memory utilization during routing table lookup, to transfer packets to the appropriate Ethernet port. Throughout this paper the point of primary concentration is packet-forwarding function of the router.

Trie based solutions are the standard approaches used today in router. However, though they are time efficient they consume a lot of memory for storage. Hence, a few approaches like [10] have been proposed to better storage and time complexity. Our approach is based on the LC-trie approach for compressing the trie. We use the same compression technique as in LC-trie. However, we try to avoid additional storage (LC-Trie uses additional data structures like base, prefix and nexthop vector to store prefixes) and processing (like backtracking), which is one of the major flaws of LC-Trie approach. Figure 4 represents the tree corresponding to Table 1 for the proposed approach. From the Figure 4 we see that unlike the LC-trie approach our approach stores all the prefixes either in the internal nodes or leaf nodes.

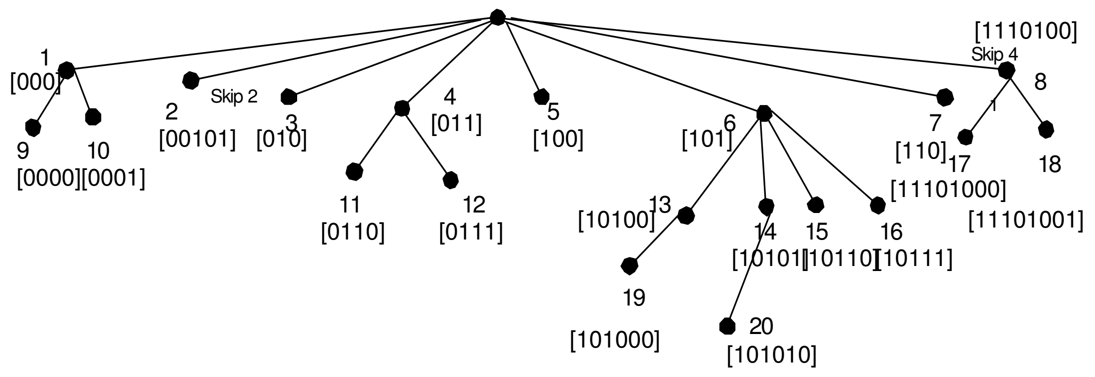


Figure 4: Scalable Time Efficient Level Compression

The algorithm first converts the routing table entries into a binary trie. Then path compression is done on this trie to reduce its depth. This path-compressed trie is now level compressed by storing the sub strings at the internal nodes and the strings at the leaves. When the routing table is built, we use a FILL FACTOR (this represents the maximum number of branches that each node can have during the build) to help make the future updates easier.

B. Storage Data Structure

Each node of our trie is represented as in Figure 5. Following is a brief description of the significance of each of the fields in the data structure.

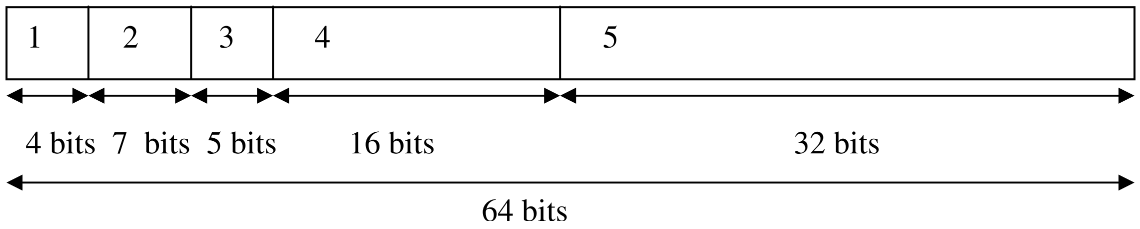


Figure 5: Proposed Data Structure for Node in Routing Table (IPv4)

Branching factor [0:3]: This indicates the number of descendents of a node. This is a 4-bit value and a maximum of 16 branches to a single node can be represented.

Skip value [4:10]: This indicates the number of bits that can be skipped in an operation. This is a 7-bit value and a maximum of 128-bit skip can be represented.
Port [11:15]: This represents the output port for the current node in case of a match. This is a 5-bit value and this field can represent a maximum of 32 output ports.
Pointer [16:31]: This is a pointer to the leftmost child in case of an internal node and NULL in case of a leaf node. This is a 16 bit value and can represent a maximum of 65536 prefixes. The current implementation assumes the number of routing table entries to be less than 65536. However, a scalable solution to consider more than 5,000,000 prefixes is discussed later.
String [32:63]: This represents the actual value of the prefix the node represents. In the current implementation, it is a 32-bit value though it can be extended to 128-bit value for IPv6 headers.

```
node = T[0]; s = testdata[k]; node = table->trie[0];
pos = GETSKIP(node);
branch = GETBRANCH(node); adr = GETADR(node); prefix=0;
result=-1; /* stored in Register */
while(branch != 0)
 {         node = table->trie[adr + EXTRACT(pos,branch,s)];
           if(pos)
                     prefix<<pos;
            if (branch)
           {
                     if(branch > 15)
                            prefix= prefix << 5| branch;
                     else if(branch > 7)
                             prefix= prefix << 4| branch;
                     else if(branch > 3)
                             prefix= prefix << 3| branch;
                     else if(branch > 1)
                              prefix= prefix << 2| branch;
                     else
                              prefix= prefix << 1| branch; }
             if(GETSTRING(node)^prefix)
        break; /* Previous node contains largest prefix and interface
       stored in result */
             else {
                    pos = pos + branch + GETSKIP(node);
                    branch = GETBRANCH(node);
                    adr = GETADR(node);
                    result = GETPORT(node);
              }
}
```

Figure 6: Algorithm for Modified LC-Trie Approach

C. Algorithm

The search algorithm (Figure 6) forms the bottleneck of the entire routing process and hence this needs to be designed very efficiently. Algorithm discussed below is used to search a string *s* in the routing table.

EXTRACT (p,b,s) is used to search *s* in the routing table, where *b* is the number of bits starting at position *p*. Let the array representing the tree be T. The root is stored in T[0].

Each entry in Table 3 represents a node in the proposed approach, for routing table described in Table 3, with the corresponding branch, skip and pointer values. In addition to these three fields each node also has a 32-bit (IPv4) prefix represented by the node, which is not indicated in the table.

Table 3: Array Representation of Modified LC-Trie

	Branch	Skip	pointer		branch	skip	pointer
0	3	0	1	10	0	0	0
1	1	0	9	11	0	0	0
2	0	2	0	12	0	0	0
3	0	0	0	13	1	0	19
4	1	0	11	14	0	0	20
5	0	0	0	15	0	0	0
6	2	0	13	16	0	0	0
7	0	0	12	17	0	0	0
8	1	4	17	18	0	0	0
9	0	0	0	19	0	0	0
10	0	0	0	20	0	0	0

1. Working of the Algorithm

The algorithm is illustrated with an example. Consider the input string 101001. We start from root node number 0. We see that the branching factor is 3 and skip value is 0 and hence extract 1st 3 bits from the search string. These 3 bits represent a value of 5, which is added to the pointer, leading to position 6 in the array. At this node the branching value is 2 and the skip value is 0 and hence we extract the next 2 bits. They have the value 0. However, we check if the string (101) matches the prefix (101). Since it matches the search continues further. We now add the value of 0 to the pointer and arrive at position 13. At this node the branching factor is 1 and skip value is 0. They have a value of 1. We again compute to see if the string (10100) is same the prefix (10100). Since it matches, we continue and add the value of 1 to 19 to obtain the pointer 20. Now see that this node represents a leaf node since the branching factor is 0. We now check to see if the string (101001) matches the prefix (101011). Since they don't match we use the previous value of the output port from the register to route the packet. The Figure 7 represents the path taken during the search.

We see that this approach does not traverse the entire trie in case the string is not present. Also, there is no separate storage for the prefixes as in case of LC-Trie. This approach checks for the match in the string at every step. However, this is not computation intensive since the string to compare is already present in the cache and a *xor* on the bits could give us the result of comparison. In the LC-trie approach, after we traverse through the trie we perform a check for the string match in the base vector, which uses hashing technique, consuming at least one memory fetch. If there is a mismatch, a check is done again on the prefix table and this requires hashing to check for a prefix match, which again requires another memory fetch. Thus compared to the LC-trie we save atleast two memory cycles for every routing lookup performed.

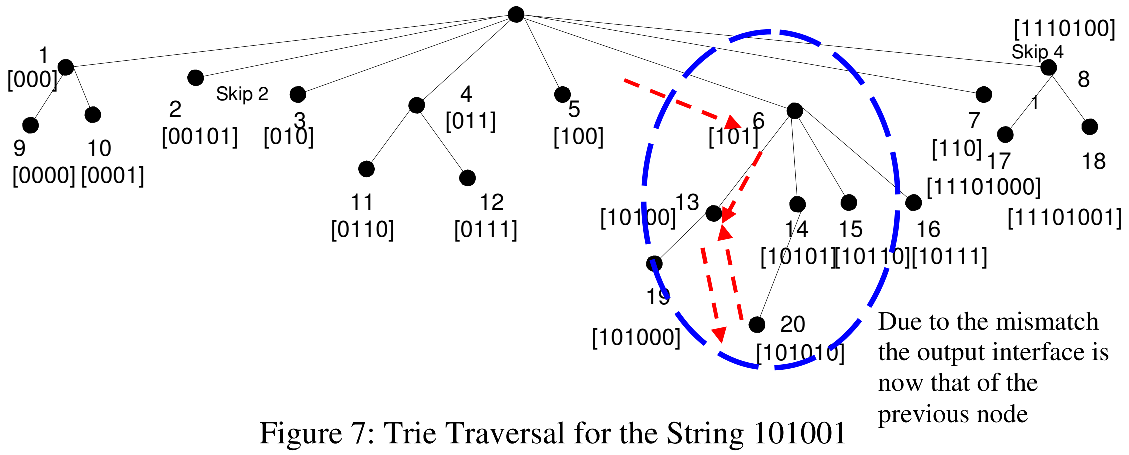


Figure 7: Trie Traversal for the String 101001

D. IPv6 Compatibility

The algorithm can easily be extended to IPv6 and to allow a maximum of 2^{37} entries. This ensures that the proposed data structure can support routing entries beyond 2005. This data structure also allows handling of a maximum of 1024 interfaces. The data structure for each node is described in the Figure 8.

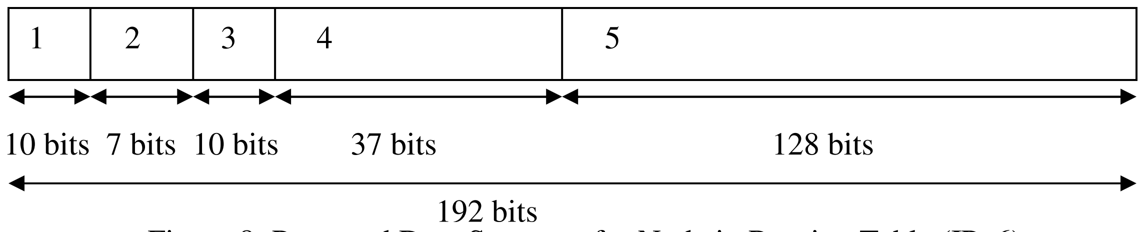


Figure 8: Proposed Data Structure for Node in Routing Table (IPv6)

E. Simulation Bench and Experimental Setup

To test and verify our approach with the LC-Trie approach we have modified the test bed used by the authors of [10]. The features of this modified test bed are as follows:

- It reads routing data from the routing table file, which is in a predefined format as discussed in the paper [10]. The routing file is an exhaustive list of routing entries (65536 entries for 16-bit pointer value).
- The algorithm can be run over a number of times by specifying *n* as a command line argument. As the number of iterations increase, it gives a good estimate of the parameters under comparison.
- Quick sort algorithm is used to sort the routing table entries.
- We have also used two different approaches to compare the performance. One uses a function call to the search algorithm and the other is an inline function. However we have used the inline function results for our comparison.

In our implementation we have used routing tables similar to that provided by the Internet Performance Measurement and Analysis project [13]. In order to compare the modified technique with LC-Trie approach the traffic was simulated and we used a random permutation of all possible entries in the routing table. The time measurements have been performed on sequences of lookup operations, where each lookup includes fetching the address from the array, performing the routing table lookup, accessing the next hop table and assigning the result to a volatile variable.

Some of the entries in the routing tables contain multiple next hops. In such cases we select the first one listed as the next hop address for the routing table, since we only consider one next hop address per entry in the routing table. However, for entries that didn't contain a next hop address a special address that is different from the ones found in routing table was used.

The following equations were used in the computation of average and standard deviation of the samples (t_i).

Average Time (avg) = t_i/n

Std Deviation (std) = $(t_i^2 - n*avg*avg)^{1/2} /(n-1)$

1. Parameters Analyzed

We have analyzed the effectiveness of our approach and compare our approach with the LC-Trie approach with respect to timing, storage and power consumption. The parameters considered in each of the cases are described below.

a. Timing

a.1. Building

Time taken to Build Routing table (B_t): This is the time taken for the algorithm to retrieve all the data from the Routing table file, sort them and build them with appropriate entries for future referencing.

Time taken to build next hop table (N_t): This is the time taken to compute all the next hop addresses from the routing table data.

a.2. Sorting

Time taken to Sort the entries (S_t): Based on a seed value the routing table entries are stored in a temporary data structure in a random fashion, which is then sorted for building the routing table.

a.3. Searching

Function Search: Time taken to search the string (using call to a function) based on *n* iterations.

F_{min}: Minimum time taken to search the string using function call.

F_{avg}: Average time taken to search the string using function call.

F_{std}: Standard deviation of the times for searching a string using function call.

F_{lps}: Average number of lookups/second using function call.

Inline Search: Time taken to search the string (using an inline function) based on *n* iterations.

I_{min}: Minimum time taken to search the string using Inline function call.

I_{avg}: Average time taken to search the string using Inline function call.

I_{std} : Standard deviation of the times for searching a string using Inline function call.

I_{lps}: Average number of lookups/second using inline function call.

b. Memory Utilization

B_m: Memory utilization in bytes for the base vector.

P_m: Memory utilization in bytes for prefix vector.

N_m: Memory utilization in bytes for next hop vector.

Trie (T_m): Memory utilization for Trie.

F. Results

Following are the results for the comparison of LC-Trie approach and proposed approach with a fill factor of 0.5 and a fixed branch at root (16). We have run this algorithm 100 times to get a good estimate of the values. This was run on an Intel Pentium II processor, 400 MHz and 256 MB RAM. The programs were written in C and complied with gcc compiler using optimization level –04.

From Table 4 we observe that time taken to build the trie is reduced by 0.14 seconds. This is mainly due to the fact that no additional computation is required to build the base and prefix vector. Also, there is no overhead of building the next hop table. The simulation results show that the proposed approach works 3.28 times better than LC-Trie approach when the prefix search is implemented as a function search and 4.11 times better when implemented as an inline function. Thus, for above mentioned system configuration we were able to achieve a lookup of approximately 6.6 Mpps in the average case.

From Table 5 we observe that the proposed approach avoids the storage for base, prefix and nexthop vector and hence occupies 2.38 times lesser storage. Though the reduction in storage for the nexthop vector is not significantly high, the storage for the base and prefix vector is greatly reduced.

The processing power savings for the two approaches were compared using an implementation based on reconfigurable processor architecture from *Tensilica* (16/24 bit Xtensa ISA, 200 MHz, 0.18 um technology, 0.7 mm^2 core area, 0.8mW/MHz core power dissipation). The Routing table used in our power analysis is described in Table 2. The result obtained is an estimate of power for one iteration. The results show that proposed approach and the LC-Trie approach consumes 4.615mW and 4.755mW for 20 lookups respectively (Table 6). This is a reduction of 0.14mW for 20 lookups. This is directly related to the fact that the time for lookup is less. The routing entries in our simulations are assumed to be stored in the DRAM and the storage power corresponding to that was computed for both the approaches. Since the storage requirements are reduced by factor of 2.38, it is expected that power consumptions will be less by that amount.

Table 4: Timing

Parameters	LC-Trie	Proposed
B_t	0.57 sec	0.43 sec
N_t	0.05 sec	0 sec
S_t	0.37 sec	0.36 sec
F_{min}	5.01 sec	1.53 sec
F_{avg}	5.02 sec	1.53625 sec
F_{std}	0.01	0.0074402
F_{lps}	1308104	4283399
I_{min}	4.12 sec	1.0 sec
I_{avg}	4.12 sec	1.005 sec
I_{std}	0.01	0.0053452
I_{lps}	1590680	6553600

Table 5: Memory Utilization

Parameters	LC-Trie (bytes)	Proposed
B_m	32769*16	0
P_m	32767*14	0
N_m	4*16	0
T_m	65537*4	65537*8

Table 6: Power Consumed

Approach	Number of	Processor	Power
LC Trie	5944854	0.8mW/Mhz	4.755 mW/20
Proposed	5769630	0.8mW/Mhz	4.615 mW/20

III. TCAM BASED ROUTER ARCHITECTURE FOR IP LOOKUP

Ternary Content Addressable Memories (TCAMs) have been emerging as a popular device in designing routers for packet forwarding and classifications. Despite their promise on high-throughput, the use of large TCAM array is prohibitive due to its excessive power consumption and lack of scalable design schemes. This section presents a TCAM-based router architecture that is energy and storage efficient. A new prefix aggregation and expansion technique to compact the effective TCAM size in a router is introduced. Pipelined and paging schemes are employed in the architecture to activate a limited number of entries in the TCAM array during an IP lookup. The new architecture provides low power, fast incremental updating, and fast table look-up. Heuristic algorithms for page filling, fast prefix update, and memory management are also provided. The empirical models for memory requirements, energy consumption and access time have been derived to estimate the performance. Results have been illustrated with two real-life large routers (bbnplanet and attcanada) to demonstrate the effectiveness of our approach.

A. Introduction

Internet protocol (IP) lookup forms a bottleneck in packet forwarding in modern IP routers because the lookup speed is unable to catch up with the increase in link bandwidth. The Ternary Content Addressable Memories (TCAMs) have been emerging as viable devices for designing high throughput forwarding engines on routers. They store don’t care states in addition to 0s and 1s, and search the data (IP address) in a single clock cycle. This property makes TCAMs particularly attractive for packet forwarding and classifications. Despite these advantages, the use of large TCAM arrays is prohibitive due to large power consumptions and lack of scalable design schemes.

The high density TCAMs, available in the market today, consumes 12-15W/chip when the entire memory is enabled. In order to support the IP prefixes that are increasing super-linearly, vendors use 4 to 8 TCAM chips. Additionally chips would also be required to handle filtering and packet classification. The power consumption resulting in using large number of chips not only increases the cooling costs but also limits the router design to fewer ports. Recent research in reducing the power consumption of the TCAM has been discussed in [14][15]. Our analysis from the CACTI-3.0 model [16] indicates that that power consumption increases linearly with the increase in the number of entries and bits in a TCAM. The TCAM look-up time increases exponentially with the number of entries. Hence, the techniques proposed for compacting the routing table will result in substantial reduction in power and look-up delay. Liu [17] has presented a novel technique to eliminate the redundancies in the routing table. However, we show that this technique takes excessive time for updating because this is based on Espresso-II minimization algorithm, whose complexity increases exponentially with the size of the original routing table. Thus the main motivation of our paper is to come up with a TCAM router architecture that consumes low power and is suitable for incremental updating needed in modern IP routers. Additionally, we minimize the memory size and look-up delay.

We propose a pipelined architecture that can achieve the above goals through further compaction of the active region of the routing table in TCAM architecture. Our paper introduces the idea of *prefix aggregation* and *prefix expansion* for TCAM, which can reduce the number of entries for comparison in a TCAM. Together with the prefix aggregation and overlapping properties, we select a limited number of pages during IP look-up instead of energizing the entire modules. The present TCAM vendors provide mechanisms to enable a chunk of TCAM, much smaller in size compared to the entire TCAM. We exploit this technology to achieve an upper bound on the power consumption in the TCAM based router. Based on statistical analysis of the current IP routing tables, we present a 2-level paged TCAM architecture. Detailed analysis and design tradeoffs are presented by varying the sub-prefix length. We also propose page filling heuristic to improve memory utilization due to paging. The concept of *bucket*-TCAM has been used to isolate storing of rarely incoming IP prefixes and to find optimal page size in the proposed architecture. An efficient memory management scheme is presented for updating the routing entries. Finally, we derive empirical equations for memory requirement and power consumption for the proposed architecture.

This paper presents results of delay and power analysis of a TCAM by modifying the CACTI III model [16], which serves as verification of our empirical equations. The access time for TCAM-based router has been addressed for the first time in this work. Case studies were made using the statistics from bbnplanet and attcanada routers to demonstrate the effectiveness of the proposed TCAM architecture. It is shown that less than 1% of power is consumed when compared to energizing the full-size TCAM implementation. Considerable saving is also noticed in memory size and look-up delay.

The paper makes the following significant contribution.

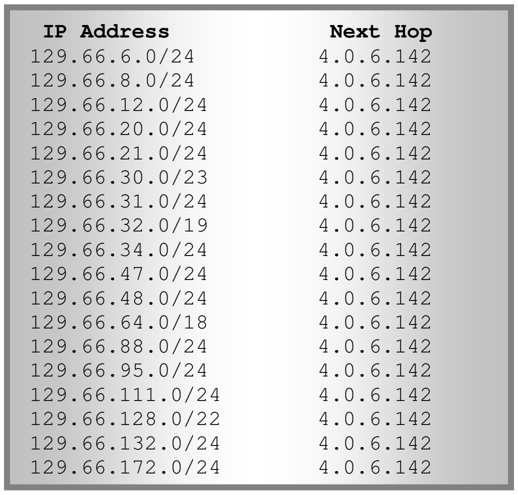
- It introduces new techniques for aggregating prefixes based on which substantial reduction in TCAM size can be obtained.
- It presents a new 2-level pipelined architecture based on which selected pages and modules can be activated during a table look-up process.
- Efficient paging and memory management techniques are derived for improving TCAM utilization and update operations.
- Analytical and simulation results are presented to show substantial gain in power, memory size and look-up delay by using the proposed architecture.

B. Prefix Compaction

The purpose of delving into the prefix properties is three fold: 1) To further increase the compaction of the routing table in addition to using the existing techniques. 2) To come up with an upper bound on minimization time lest it become a bottleneck in the routing lookup. 3) To derive an upper bound on the power consumption during a lookup process. The previous attempts to reduce the routing table size in TCAM using prefix properties have been achieved using the property of *Pruning* and *Mask extension* [17]. *Pruning* achieves compaction by storing only one of the many overlapping prefixes that have the same next-hop. Two prefixes are overlapping if one is an enclosure of the other. Let P_i denote the prefix P_i and $|P_i|$ denotes the length of prefix P_i then $P_i \in P$ is called *enclosure* of P_j, if $P_j \in P$, $j \neq i$ and $|P_i| < |P_j|$, such that P_i is a sub-prefix of P_j. If P_i and P_j have the same next hop then they can be represented by P_i. Thus, a set of overlapping prefixes $\{P_1,P_2,P_3,..\ P_n\}$, such that $|P_1|<|P_2|<|P_3|..<|P_n|$, having the same next hop can be replaced with a single entry P_1. If an update deletes the entry P_1, the set of overlapping prefixes $\{P_1,P_{2,}..P_{in}\}$ should be represented by the entry P_2. When an update adds a new entry P_i, such that $|P_i|<|P_1|<|P_2|..<|P_n|$, then the existing entry P_1 is replaced with P_i. However, if the new entry P_i arrives, such that, $|P_i|>|P_1|$, then no changes are made to the routing table, except for the updating of set of overlapping prefixes.

The *Mask extension* property logically minimizes two or more prefixes to a minimal set, if these prefixes have the same next hop. The logic minimization is an NP-complete problem and has been addressed using the Espresso-II algorithm [18]. Using Espresso-II, the prefixes $\{P_1, P_2, P_3 ... P_n\}$ are minimized to $\{P'_1, P'_2, P'_3, .. P'_m\}$, such that $m \leq n$.

While the *Pruning* and *Mask extension* technique together compacts about 30-45% of the routing table, one need to investigate the overhead in using these approaches for real time updates. The time taken for pruning is bounded and is independent of the size of the router. However the logic minimization algorithm using Espresso-II has an exponential runtime with input size. Thus based on the technique discussed in [17], the input to Espresso-II algorithm can be as high as 15,146 for the attcanada router. The runtime for such a large size input data can take several tens of minutes and is very expensive for incremental updates. We tackle this problem by having an upper bound on the input to the minimization algorithm, which is independent of the router size, using another property called *prefix aggregation*. The minimization of the prefixes based on this property is time bounded.



IP Address	Next Hop
129.66.6.0/24	4.0.6.142
129.66.8.0/24	4.0.6.142
129.66.12.0/24	4.0.6.142
129.66.20.0/24	4.0.6.142
129.66.21.0/24	4.0.6.142
129.66.30.0/23	4.0.6.142
129.66.31.0/24	4.0.6.142
129.66.32.0/19	4.0.6.142
129.66.34.0/24	4.0.6.142
129.66.47.0/24	4.0.6.142
129.66.48.0/24	4.0.6.142
129.66.64.0/18	4.0.6.142
129.66.88.0/24	4.0.6.142
129.66.95.0/24	4.0.6.142
129.66.111.0/24	4.0.6.142
129.66.128.0/22	4.0.6.142
129.66.132.0/24	4.0.6.142
129.66.172.0/24	4.0.6.142

Figure 9: Sample Trace of Routing Table from `bbnplanet`

1. Prefix Aggregation

In this sub section, we introduce a new property of the prefixes called *prefix aggregation*. Figure 9 represents a portion of the routing table dump taken from the bbnplanet router. It represents all the prefixes in the routing table starting with 129.66 and having prefix length > 16 and ≤ 24. We call 129.66, the largest common sub-prefix (LCS) for the set of prefixes. The LCS for any prefix is the sub-prefix whose length is the nearest multiple of 8 such that $|S_i|<|P_i|$, where S_i is the LCS of prefix P_i ($LCS(P_i)$). However if the prefixes under consideration are such that $|P_i| > w_1 \ \forall \ P_i$, where w_1 is an integer, then LCS of P_i is $(p_{i1} p_{i2} ... p_{iw1})$ if $|P_i| \leq (w_1/8 + 1)*8$ (p_{ik} represents the k^th bit of the prefix P_i). From the statistics collected, we observe that if the prefixes are grouped based on their LCS, their next-hop is mostly the same with a few exceptions. Interestingly, it is also observed that prefixes having different maximum common sub-prefix usually do not have the same next-hop. According to these observations, we may state that the degree of compaction achieved by independently applying minimization on each set of prefixes with same largest common sub-prefix, is nearly equal to the minimization achieved when they are considered together.

Table 7 gives an indication about the extent of compaction achieved by prefix aggregation and the maximum compaction possible in the two routers. Thus based on the prefix statistics in the core routers, we can presume that the set of largest common sub-prefixes could be treated as mutually exclusive. Notice that our algorithm does not produce maximum prefix compaction, but it saves considerable compaction time by limiting the inputs to the Espresso algorithm, used for such compaction. As a result, the technique is applicable for incremental (on-line) updating as opposed to the technique in [17]. More detailed explanation on this property is given later in this section.

The property of *prefix aggregation* directly gives an upper bound on the number of possible prefixes that can be given as input to the minimization algorithm, without significantly affecting the total amount of compaction. This input set is the set of prefixes with maximum possible size such that each prefix in the set has the same LCS. Let the largest common sub-prefix for prefix $P_i = (p_{i1} p_{i2} ... p_{i|P_i|})$ be represented as $S = (p_{i1} p_{i2} ... p_{i((|P_i|-1)/8)*8})$. Then the aggregated set P is a set of prefixes $\forall P_i \in P$ such that the maximum common sub-prefix of P_i is S.

Lemma II.1: The maximum number of prefixes with the same largest common sub-prefix in a prefix set is 256.

Proof: Let S be the largest common sub-prefix for the set of prefixes P_i. The Classless Inter-domain Routing (CIDR) addressing was introduced so that if S covers all ranges of prefixes P_i, such that $LCS(P_i) = S$ and, $|S| < |P_i| \leq (|S|+8)$, then new subnets can be added with prefix Pi, that will result in networks having smaller number of hosts. Thus to find the maximum possible number of such prefixes that can be added, we assign all possible combinations of P_i, such that

$|Pi| = (|S|+8)$. This will ensure that it is not possible to add any Pi such that $|S| < |P_i| < (|S|+8)$. Thus we can have a total of $2^{((|S|+8) - |S|)} = 256$ combinations.

Table 7: Comparison of Prefix Minimization Using Prefix Aggregation Property in `attcanada` and `bbnplanet` Router

Router	Total # of prefixes	Maximum Possible Minimization # of prefixes	Prefix Aggregation Minimization # of prefixes
AT&T-Canada	112412		57837
Bbn-planet	124538	69646	71500

2. Prefix Expansion

The concept of prefix expansion for IP lookup has been discussed using software approaches like [19]. We adopt this property to further compact the routing table. The Prefix Expansion property can be represented mathematically as follows. If P_i represent a prefix, such that $| P_i|$ is not a multiple of 8, then the prefix expansion property expands P_i to $P_i.X^m$ such that, X=don't care and $m = 8-(|P_i| \bmod 8)$. The operator "." represents the concatenation operation.

From Lemma II.1 we know that the maximum input size given to the minimization algorithm for bounded runtime is the set that has maximum number of prefixes (256) having same largest common sub-prefix. For each prefix $P_i \in P$, that has a largest common sub-prefix S, $|P_i|$ may not be a multiple of 8. However Espresso-II algorithm will provide more compaction if the entire P_i's are of same length. Hence, to increase the compaction, we expand the prefix so that its length is a multiple of 8, and give it as input to the Espresso-II algorithm.

The run time of the Espresso-II algorithm is not only bounded by the number of inputs but also by the bit length of the input. From previous discussions we observe that the input set to the Espresso algorithm is the set of prefixes that have the same LCS. Thus, the largest common prefix that is a part of all the prefixes is redundant to the Espresso-II algorithm. However it would be useful, to only give the varying bits of the set containing the largest common sub-prefix, which is the least significant 8 bits of each prefix after prefix expansion. Thus for each $P_i \in P$ that has the same LCS S, the optimal input set of prefixes to the Espresso algorithm to maximize the compaction with an upper bound on runtime is the set P',

such that $\forall P'_i \in P'$

$N. X^q_i$ for $|P_i| \neq m*8$, m is an integer

N for $|P_i| = m*8$, m is an integer

where,

$N=(p_{i_k} p_{i_{k+1}} ... p_{i_{|P_i|}})$

$k=8*((|P_i|-1)/8)+1$

$q_i = 8-(|P_i|)$

3. Overlapping Prefixes for an IP Address

The maximum number of overlapping prefixes for a given IP address indicates the minimum number of prefixes one needs to search during a lookup operation. Thus, if we search a bounded numbers of prefixes during any lookup operation, the power consumed will be bounded (This is based on the assumption that only the entries that are enabled during the lookup operation contribute to the power consumption). In the following section, we show that during a search process, only a bounded number of prefixes (256*3) will be compared. These prefixes will contain the set of overlapping prefix for any IP address.

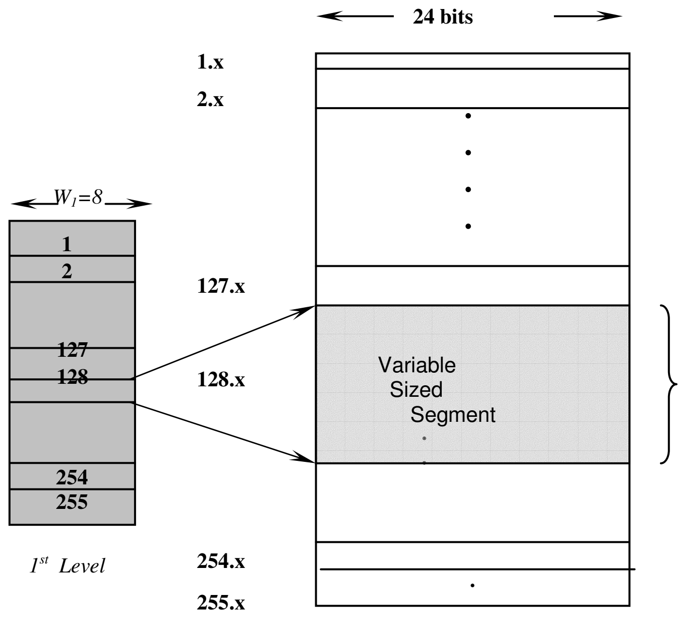


Figure 10: Segmented Architecture for Routing Lookup Using TCAM

Lemma II.2: The total number of overlapping prefix for a given IP address is ≤ 25.

Proof: Consider the incoming IP address I = (i_1i_2 .. i_{32}). For every I, we can generate a P_i/l, such that, $1 \leq l \leq 32$, and P_i = ($i_1 i_2 ... i_l$). Since there are 32 such values of l, we can have 32 such possible overlapping prefixes of I. Since in today's routers there are no prefixes of length < 8, the maximum number of possible overlapping prefixes for I is 32-7=25.

C. Deriving a TCAM Architecture

In the architecture, compaction of the routing table is achieved by exploiting the techniques developed in this section. Notice that our approach has been different from [17] to accommodate fast incremental updates. We adopt a 2-level routing lookup architecture (shown in Figure 10) with w_l bits in the 1st level. The 1st level contains w_l-bit sub-prefix which is compared with w_l most significant bits of incoming IP address. If there is a match in the 1st level, it enables corresponding region in the 2nd level. The size of this selected region varies from router to router. This region is called as segment in Figure 2. This would mean that the worst-case power consumption is decided by the size of the largest segment, which is not bounded. For large sized routers the segment size could be very large. For example, with w_l = 8, the size of the largest segment in bbnplanet router is 7580 entries.

Choice of the number of bits in the 1st level affects the compaction and performance of the proposed architecture. As a requirement for our architecture, the entries in the 1st level TCAM do not store any don't care states. This will ensure that only a unique segment in the 2nd level will be selected. Depending on the value of w_1 at the 1st level, the size of the TCAM arrays and total memory requirements will vary and hence the performance.

The compaction technique first selects all the prefixes that are greater than w_1 bits from the routing table traces. Using the property of prefix overlapping we remove all the redundant entries in the routing table. We then use the property of prefix aggregation to form sets of prefixes having largest common sub-prefix. Each of these sets are expanded using the prefix expansion technique and are given as input to the Espresso-II minimization algorithm. Then minimized set of prefixes is stored in the 2nd level TCAM that are of length $32-w_1$.

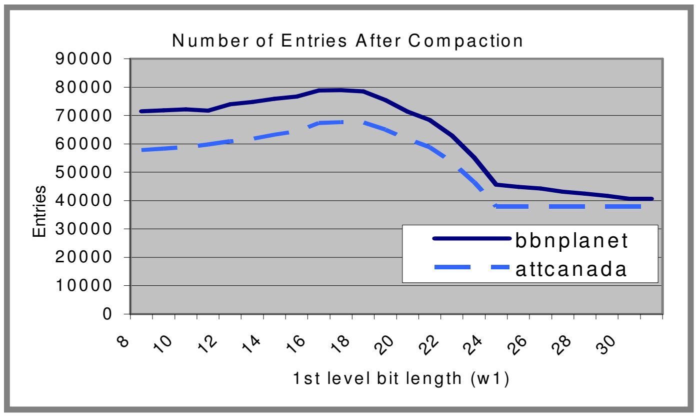


Figure 11: Total Number of Entries after Compaction

Figure 11 shows the number of entries in the TCAM after compaction for `bbnplanet` and `attcanada` routers, using the compaction technique while varying the value of w_1. For the value of w_1 between 8 and 18, the total compaction decreases with increasing w_1 for both the routers. This is because as w_1 increases the input to the Espresso-II algorithm decreases (prefixes $< w_1$ are not given as input to the Espresso algorithm). However for $w_1 > 18$, we see that the number of compacted entries is decreasing with increasing w_1. This is because of the fact that the number of prefixes available for compaction gets reduced in the 2nd level TCAM. At this point the benefit of compaction is not noticeable.

In this sub section we present the detailed architecture of the TCAM based packet-forwarding architecture. The discussions on various components of the architecture and design parameters are presented. We introduce the heuristics based on prefix properties to obtain a bound on power consumption based on the active TCAM entries for a single lookup operation. Heuristics to store the pages efficiently in the TCAM for high memory utilization is also discussed. Lastly, we also introduce an empirical power and memory requirement model based on the parameters involved in the architectural design.

D. Paged TCAM Architecture

One of our goals is to estimate the total memory requirements in TCAM arrays and maximum power consumption in the lookup process for different values of prefix lengths (w_1). In order to determine the maximum power consumption per IP lookup, we need to determine maximum number of TCAM entries enabled during any lookup process. These estimations depend on the design details of the proposed architecture.

The TCAM router with two-level lookup arrangements has been depicted in Fig.3. Each entry in the 1st level contains w_1-bit sub-prefix which is compared with w_1 most significant bits of incoming IP address. If there is a match in the 1st level, it enables corresponding region in the 2nd level. The size of this selected region varies from router to router. This region is called as segment in Figure 3. This would mean that the worst-case power consumption is decided by the size of the largest segment, which is not bounded. For large sized routers the segment size could be very large. For example, with $w_1 = 8$, the size of the largest segment in `bbnplanet` router is 7580 entries.

The implementation scheme to realize segmentation in TCAM array is non-trivial. In order to implement the segmentation concept, we consider a paging scheme for actual implementation of the segments. Though the paging scheme for TCAMs are also used in [14] and [15], we handle the paging differently due to architectural differences.

We now discuss heuristics to efficiently store enabling bounded number of entries for power consumption.

1. Determining Bound on Page Size

Here, we consider selecting a bounded number of pages in a segment during IP lookup instead of selecting all the pages using *prefix aggregation* technique. The prefixes having the same largest common sub-prefix are stored in the pages that have the same page ID's, which are represented by their LCS values. Then the maximum number of pages enabled during the lookup process is bounded (Lemma III.1). In the bbnplanet router, the number of entries in each page can range from minimum of 1 to the maximum of 256. Though this technique can achieve a bound on the number of entries in a page, it may not be an elegant solution due to its low memory utilizations.

Lemma III.1: When the pages are created using aggregation technique, the maximum number of pages enabled that contains the set of overlapping prefixes during a lookup process is ≤ 3.

Proof: From Lemma II.2 we know that the total number of possible overlapping prefixes for a given IP address is ≤ 25. One needs to find out the maximum number of pages that will be searched based on the aggregation technique. This is the set of pages containing the set of overlapping prefixes. Since we have considered the number of bit-lines in the 1st level (w_1) ≥ 8, the total number of overlapping prefixes is ≤ 24. We now prove the bound for the value of w_1=8, which has the largest number of overlapping prefixes. The prefix aggregation technique groups the prefixes such that they have a common prefix, which is a multiple of 8 for w_1=8. Hence there are three such groups possible with maximum common prefix lengths of 8, 16, and 24. Thus the total number of pages containing overlapping prefixes cannot exceed 3 based on the aggregation technique.

Corollary: The upper bound on the number of entries that need to be searched during an IP lookup cannot exceed 256*3.

The statistics from the bbnplanet router shows that the segment sizes vary depending on the value of w_1. Thus if the page size is large, having separate pages for the smaller sized segments would result in a wastage of space. Similarly if the segment sizes are large having small sized pages will increase the number of page ID's and hence the memory utilization. Hence one needs to have a technique to compute the right page size depending on w_1 that would minimize the memory consumption. In the next section we describe the heuristics to obtain the optimal page size, among all the possible page sizes that are possible. Since the sets formed using the aggregation technique cannot exceed 256 entries, the maximum size of the page cannot exceed 256 entries. We refer to the term β to refer to the page size. Our goal is to find the optimal value of β (β_{w1}) for all possible w_1, which will optimizes memory.

2. Page Filling Heuristics

Though the aggregation technique gives an upper bound on the power consumption, it may increase the TCAM's size due to memory under utilization. We present a heuristics based on the maximal covering of prefixes to increase memory utilization. Let a *cube* represent a single entry for a set of prefixes and *covering* represents the set of cubes that cover all the prefixes that have the same LCS(P_i). Our goal is to find the minimal set of such *cubes* that combine prefixes P_i having the same $|S_i|$. Since $|S_i|$ can have a maximum of 3 values for any value of w_1 in the proposed architecture, there can be a maximum of 3 overlapping *cubes* for a given IP address if the *cubes* in each *cover* are made non-overlapping. This ensures that the maximum number of active or enabled entries cannot exceed 256*3. This will be true if all the prefixes in the *cube* C_i can be arranged in pages so that the total number of entries in all pages does not exceed 256. Each of these cubes will represent the page ID for the page containing the prefixes it overlaps. We also introduce a parameter γ, called *fill factor*, which represents the fraction of page filled during reconfiguration process that covers prefixes having different LCS values. The page-filling algorithm is presented Figure 12 (a) & (b).

```
StorePage(Cᵢ,P,β,γ)
While P ≠ 0
  Create New Page with Page ID Cᵢ
Entry=0
While Entry < β*γ
      AddToPage(Pᵢ)
      P= P-{Pi}
      Entry++
EndWhile
EndWhile
```

Figure 12(a): Heuristics for Storing Prefixes

The page-filling algorithm tries to fill the entries into the pages, each of size β, efficiently by trying to find the minimal number of cubes that are non-overlapping and cover all the prefixes using MinimalCoverSet. The StorePage algorithm ensures that no page has more than β*γ entries.

```
FillPages(P,$w_1$,β,γ,C')
//$P_i \in P$, such that $|P_i| > w_1, \forall P_i$
Let $Q_{w1} = p_{i1}p_{i2}p \ldots p_{iw1}$
$C_{max}$=0
  For all $P_i \in P$ with same |LCS($P_i$)|
  and $Q_{w1}$
  For
  |LCS(Pi)|<=l<=(|LCS(Pi)|/8+1)*8
           Ci=MinimalCoverSet(P, γ)
      EndFor
      For all $P_i \in P$ covered by $C_i$
      If(Ci covers Pi's having same
LCS(Pi))
          StorePage( Ci,P,β,γ)
    Else
          StorePage( Ci,P,β,1)
      EndFor
      PageTable_$Q_{w1}$ += |Ci|
      If(PageTable_$Q_{w1}$ > $C_{max}$)
           $C_{max}$=C
      EndFor
EndFor
Return PageTable_$Q_{w1}$, $C_{max}$
```

Figure 12 (b): Page Filling Heuristics

E. Comparator, Page Tables, Pages and Bucket

Figure 13 represents detailed architecture of the proposed approach. The 1st level lookup consists of comparators and page tables, and the 2nd level lookup consists of pages and the bucket. The concepts of page tables and buckets are described later in this section. When an IP address is looked up using the forwarding engine, depending on the *cube* that covers the IP address, the comparator enables the appropriate page table. The 32-bit IP address is again looked up in the page table, and if there is a match all the pages covered by the prefix are enabled. The least significant 32-w_1 bits of the IP address are then looked up in the pages to find the longest prefix match. However if there is no match in the page table the 32 bit IP address is looked up in the bucket for a match. We now give a detailed description of each of the architectural components.

The page tables store the page ID's associated with the pages. These too are implemented using TCAMs. Page tables can be thought of as pages with word length of 32 bits. The word length of page tables is 32-bit length because the maximum length of the *cube* can be as high as 32 bits. The minimum size of the page table is given by $C_{max,}$ as computed in the `FillPages` algorithm. The page tables also have some empty entries (stored with all 0's, so that they don't match any address), which will be used for memory management as discussed in Section IV C.

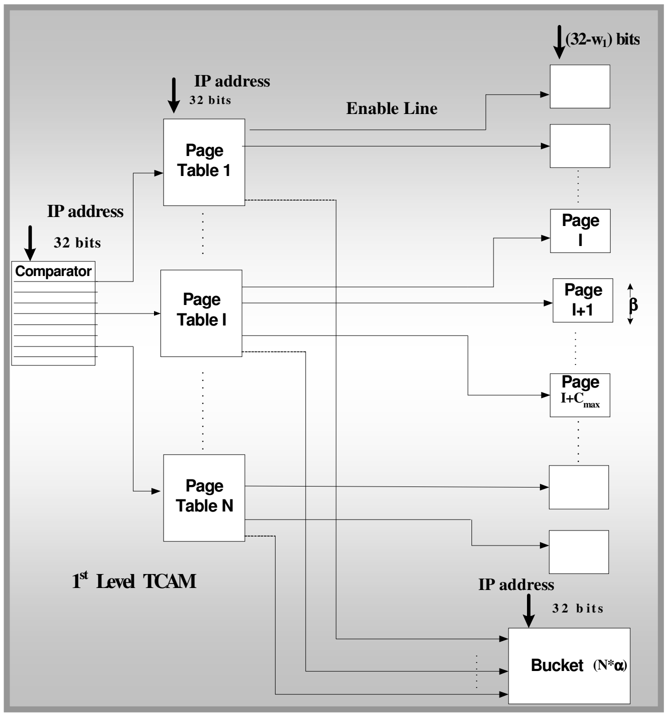


Figure 13: Detailed Architecture of EaseCAM

The range comparator is designed (using TCAM) such that it would selectively enable the TCAM page table that contains the cubes covering the incoming IP address. From the router statistics, we know that the number of page tables required is few in numbers. Thus, the number of comparators required is very small (`bbnplanet` statistics showed that the number of comparator varied from 14 to 308 for different values of w_1). Hence, the overheads due to comparators for memory usage and power consumption are negligible. The heuristics for creating the page table and comparators is shown in Figure 14. It is important to note that the solution assumes the number of entries in the page table to be small. This assumption is true for today's routers. However, theoretically the number of entries in each page table could be as high as 256*256 entries in a page table. Thus the power consumed per lookup is no longer controlled by the number of pages enabled per lookup. From the experimental results it was found that the number of entries in the pages is not greater than 64 and hence the power consumed is determined by the number of entries enabled in the pages.

```
CreatePageTable(C,w1)
//Find Cubes for different value of w1
For all Ci corresponding to $Q_{w1}$
     PageTable_$Q_{w1}$=PageTable_$Q_{w1}$+Ci
// Avoiding under-utilization
Combine all PageTable_$Q_{w1}$ ST
     (PageTable_$Q_{w1}$ < $C_{max}$)
     && Li=(Minimize(PageTable_$Q_{w1..j}$)==1)
// The newly formed cubes Li is the comparator value
     AddComparator(Li)
```

Figure 14: Algorithm to Create Page Table and Comparators

1. Building the Routing Table: Example

Figure 15(a) is a sample routing table trace taken from the core router. Figure 15(b) illustrates how the prefixes are stored based on the largest common sub-prefix. This kind of storage results in under-utilization, especially when there are not many prefixes covered by the largest common sub-prefix. Figure 15(c) illustrates the effect of aggregating prefixes that have different largest common sub-prefix. It can be seen from the figure that the architecture refines the process of prefix matching successively through multiple stages of pipelining.

IP Address	**Next Hop**
129.66.1.20/28	4.0.6.142
129.66.1.21/29	4.0.6.142
129.66.1.23/29	4.0.6.142
129.66.1.24/29	4.0.6.142
129.194.1.20/25	4.0.6.142
129.194.2.28.0/27	7.0.16.142
129.194.3.23/25	7.0.6.77
129.194.4.23/25	8.0.6.77
170.1.1.1/31	4.0.6.142
170.1.1.2/31	4.0.6.142
170.1.1.3/31	4.0.6.142
170.1.1.4/31	4.0.6.142

Figure 15(a): Routing Table Trace

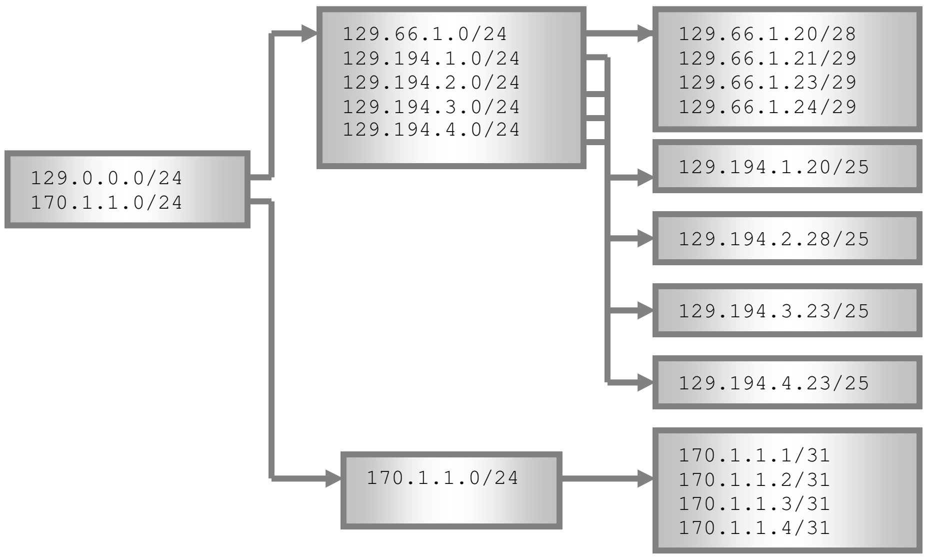


Figure 15(b): Aggregating Prefix Based on Largest Common Sub-prefix

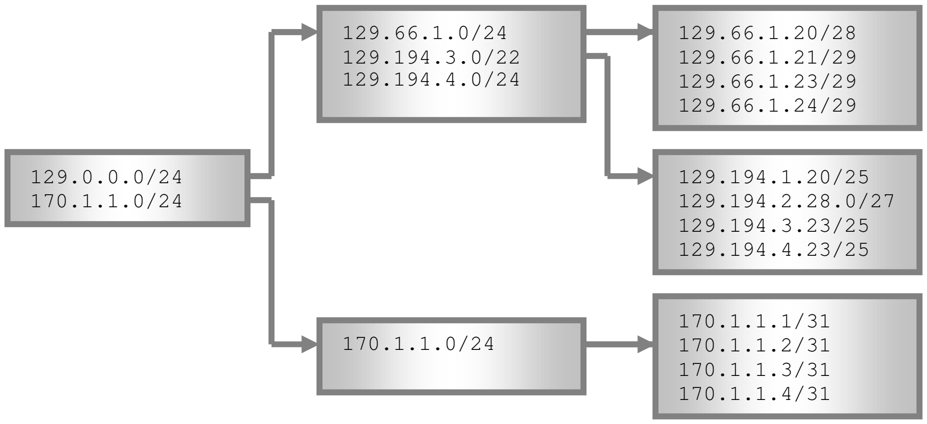


Figure 15(c): Aggregating Prefixes Having Different Largest Common Sub-prefix

2. Bucket

The proposed scheme stores all the prefixes of length $\leq w_1$ separately into a bucket that is 32 bits wide. These entries will have to be searched only when there is a mismatch in the 1st level. We now define the parameter α as the fraction of routing table entries present in the bucket. Since the bucket contains all the prefixes $\leq w_1$, the value of α increases with the increase in w_1. We will see in Section V that the value of total estimated power is mostly controlled by α for larger values of w_1 and independent of the bucket size for smaller values of w_1.

The statistics from different routers show that the number of prefixes in the bucket is very small for $w_1 \leq 16$ (approx 2% of the routing table). Since the growth rate of the prefixes of length ≤ 16 is small, it is very unlikely that the bucket will overflow when designed with reasonable *fill factor*. The fill factor for the bucket α_f is defined as the proportion of bucket that are filled with prefixes ≤ w_1 during router reconfiguration. It may be mentioned here that the bucket is not only used for storing prefixes of length ≤ w_1, but also prefixes that arrive due to an update which cannot be grouped as a part of a page in the 2nd level. These type of prefixes rarely occur and wouldn't attribute to bucket overflow for smaller values of w_1.

It is important to note that the bucket, pages and the comparators are all of 32-bit word length, while the pages in the TCAM have word length 32-w_1. Thus we cannot place them in the same TCAM chip as the prefixes. However, the total memory for the buckets, page tables and comparators will be of very small size, and they can be implemented using smaller sized TCAM chips word length 32 bits each.

```
ForEach $w_1$ from 8 to 31
//These are the possible values of $w_1$
ForEach j from 1 to 8
//$\beta$ ranges from 2 to 256, powers of 2
      $\beta$ = 1<<j
      C'=0
      Page=FillPage(P,$w_1$,$\beta$,$\gamma$,C')
      //Fill the page using heuristics
      Mem= Page* $\beta$*(32-$w_1$)/32 + Page + $M_c$
      // $M_c$ memory of comparators used
      If(Mem < $Mem_{w1}$)
      //Obtain optimal $\beta$ value
                  $Mem_{w1}$=Mem
                  $Page_{max}$=Page
                  $\beta_{w1}$= $\beta$
EndFor
EndFor
```

Figure 16: Heuristics for Finding Optimal Page

3. Optimal Page Size

Having discussed various components in the architecture that contribute to the memory requirements, we now propose a heuristics to find the optimal page size for a given w_1 so that the total memory required is minimal. The algorithm described in Figure 16, computes the optimal value of $\beta = \beta_{w1}$ for each value of w_1. The sum total of Mem_{w1} and the bucket size will give the total memory required for w_1 bits in the 1st level.

F. Empirical Model for Memory, Power and Access Time

We now present empirical models to compute the total memory requirements, energy consumption and access time for the proposed architecture. We use α as the maximum bucket size, β as the page size, γ as the fill factor of the pages and N as the number of entries in routing table after compaction. Also from the algorithm we see that β_{w1}, represents the optimal page size for $Page_{max}$ pages that minimizes memory for w_1 bits in the 1st level lookup. The minimum memory requirement due to the proposed architecture (32 bit entries equivalent) is:

$= \beta_{w1} * Page_{max} * (32-w_1)/32 + Page_{max} + Page_{max}/C_{max} + N*\alpha/ \alpha_f$ …(1)

The above equation calculates the minimum memory required for different values of w_1. We have approximated the memory used by the page tables to the number of pages used for optimal $\beta_{w1.}$ This is a reasonable approximation since the range comparators can be designed to squeeze the entire page ID's into the page tables with a negligible wastage in memory. The number of comparators used can be approximated to the number of page tables and is given by C_{max}. The bucket size is represented by $N*\alpha/ \alpha_f$, and N/ α represents the number of prefixes in the bucket during reconfiguration time. We assume that the prefixes added in the future will be very few and negligible since they represent rarely occurring prefixes. This assumption is true for small values of w_1.

The power estimation based on the number of entries enabled is not an accurate model to measure power. Due to VLSI layout and technology intricacies, the power consumption is not directly proportional to the number of bits or entries used. We have extended the CACTI-3.0 [29] model to estimate the power of TCAM, which is a standard approach for VLSI design. It may be noted that the original model was developed for cache memories. We have not described the modifications here due to lack of space. Based on this model the energy consumption in TCAM is given by:

$E(\mathit{rows},\mathit{tagbits})$ (pJ)= $(0.126284*\mathit{rows}*\mathit{tagbits} + 1.63569*\mathit{rows} + 1.97232*\mathit{tagbits} + 0.875828) / \mathit{fudge\_factor}$ …(2)

where, rows is the number of rows enabled, *tag bits* is the word length of the TCAM, and the *fudge_factor* is (0.8/0.18), which is technology dependent.

The maximum power consumption using CACTI-3.0 for a single lookup process is given by

$$\mathrm{Max}(f*E(256*3,(32-w1))\ ,\ f*E((P*N*\alpha/\alpha_f),32)+ f*E(C_{max},32)+ f*E'(Page_{max}/C_{max}\ ,32) \qquad ...(3)$$

where f is TCAM's frequency of operation.

The estimated power is due to the total entries looked up in comparators and page table in the 1st level and bucket or TCAM pages (maximum of the two) in the 2nd level. The access time can be estimated using CACTI-3.0 model similar to energy equation.

G. Incremental Update

The dynamically changing routing table could have 100s of updates per second. Though most of the updates (inserts/deletes) are route flaps, still 10s of updates per second will be required in the backbone router. Thus we need a fast incremental update algorithm for the proposed architecture.

```
Insert($P_i$)
If($|P_i| \leq w_1$)
InsertIntoBucket($P_i$)
Else
If($\exists$ $C_i$ that covers $P_i$)
        find Ci such that
        |LCS(PCover($C_i$))|= |LCS($P_i$)|
        //PCover($C_i$) returns a prefix covered by $C_i$
                P=Cover($C_i$)
                P=P+{$P_i$}
                Q=Minimize(P)
                DeleteFromTCAM($Q' \cap P$)
                InsertToTCAM($Q \cap P'$)
                UpdatePageTable($C_i$)
        Else
                InsertToBucket($P_i$)
```

Figure 17: Incremental Insertion

1. Insertion

The architectural design proposed in this paper is very conducive to the fast incremental updates. When a new prefix P_i is to be added into the TCAM, we first check if it is one of the rarely occurring prefixes. This could be done by checking if $|P_i| \leq w_1$ or if the prefix is not covered by any cube C_i that is present in the page table, and adding it to the bucket. However, if the prefix is covered by some cube C_i, there could be a maximum of 3 cubes that covers this prefix P_i. Based on the heuristics for storing the prefixes into pages, we know that the prefix must be inserted into the page whose page ID (C_i), is such that |LCS (Cover(C_i)) | = |LCS(P_i)|. Before we insert the prefix into cube C_i, we minimize the new prefix with all the prefixes P covered by C_i (Cover(C_i)), that have the same LCS as the new prefix P_i. Then the minimized set of prefixes is updated into the TCAM page with page ID C_i. We need to update the value of C_i, based on the state of the current prefixes in the page. This is done by the UpdatePageTable(C_i) procedure in the insertion heuristic (Figure 17).

2. Deletion

The deletion process is similar to the insertion process bit is simplified by the fact that the minimization is done on the raw prefix data and not on the previously minimized set. Since the number of such prefixes cannot exceed 256, it is not an overhead in terms of computation. Figure 18 shows the incremental deletion algorithm.

```
Deletion(Pi)
If(|Pi| < w1)
DeleteFromBucket(Pi)
Else
If(∃Ci that covers Pi)
     Find Ci such that
     |LCS(PCover($C_i$))|=|LCS($P_i$)|
     P=Cover(Ci)
     P=P-Pi
     Q=Minimize(P)
     DeleteFromTCAM(Q'⋂P)
     InsertToTCAM(Q⋂P')
     UpdatePageTable($C_i$)
Else
     DeleteFromBucket($P_i$)
```

Figure 18: Incremental Deletion

It is important to note that Cover(C_i) function will return the set of prefixes that have the same maximum common sub-prefix as the new entry, but without minimization being performed on them. As we have shown earlier in Lemma II.1, this will not result in minimizing more than 256 entries. The details of `InsertToTCAM`, `InsertToBucket`, `DeleteFromTCAM`, `DeleteFromBucket` and `UpdatePageTable` schemes are not provided due here to lack of space.

The runtime for the update algorithm (insertion and deletion) is bounded by the time for minimization using Espresso-II algorithm. Table 8 gives the runtime for minimizing the prefixes using the proposed approach and the technique proposed by [17]. The size in Table 8 represents the maximum input that would to be given to the Espresso-II algorithm for minimization during prefix update. The Espresso-II algorithm was run on an Athlon dual processor running 1.6 GHz, to minimize the prefixes[1] when a request for updating a new prefix is issued by the neighboring router using the approach in [17], the worst-case incremental prefix update took 63.04 sec for the `bbnplanet` router and 1098.47 sec for `attcanada` router, which are not practical values for high-performance routers. The proposed approach on the other hand took a worst-case time of 0.006sec for incremental prefix update. The value is not only small and practical, but also bounded since at any point of time the number of inputs to Espresso-II algorithm never exceeds 256. The Espresso-II algorithm's runtime also increases with the number of input bits. The proposed algorithm uses 8 bits as opposed to 32 bits in [17] and hence makes the computation time faster.). It is important to note that the functions `Cover`, `Minimize`, `DeleteFromTCAM`, `InsertToTCAM`, `UpdatePageTable`, `InsertToBucket` have a worst case complexity of O(256). Hence, the insertion and deletion scheme is incremental and bounded.

Table 8: Comparison of Incremental Update

Router	Total Prefixes	Approach in [1]		Proposed Approach	
		Size	Time (sec)	Size	Time (sec)
Attcanada	112412	15146	1098.47	223	0.005
Bbnplanet	124538	7580	63.04	256	0.006

[1] Prefixes here indicate the routing table prefixes after initial compaction (prefix overlapping and minimization)

3. Memory Management

When the router is reconfigured the comparators, page tables and pages are setup based on the optimal architecture for the router. The design of the router is based on the prefix statistics and hence it is less likely that the pages will overflow. Also the page fill factor γ ensures that during the build time of the router the pages have enough space for future updates. Also, most of the prefix updates are route flaps. So it is more likely that the same set of prefixes will be added and deleted that would result in very less chance of an overflow.

However, it is still possible that the pages could overflow. In the previous section when we discussed the insertion algorithm, we did not consider the possibility of an overflow. Hence, we present a memory management technique that will effectively reorganize the pages without affecting the update time. Let P_i be the prefix that resulted in the overflow. Then the insert algorithm with the memory management is shown below.

```
Let $Q_{w1} = p_{i1}p_{i2}p_{i3}..p_{iw1}$
For all $P_i$ such that $Q_{w1}$ covers $P_i$
    FillPages(P,$w_1$, $\beta_{max}$,1,C)
    UpdatePageTable($C_i$)
    ReprogrammeDeMux($C_i$)
```

Figure 19: Memory Management Heuristic

The memory management algorithm ensures that only the pages that have their Page ID same as C_i are recomputed during an overflow. It is important to note that if an overflow results in creating an extra page, the free pages already present in the TCAM will be used and the page tables will be updated appropriately. As we mentioned earlier some of the entries in the page tables were kept empty during reconfiguration process, which will be used during an overflow. The memory management technique assumes that the page table will not overflow. Though this is very unlikely to happen, it cannot be ruled out. In such a case we reconfigure the router all over again. The memory management heuristics is shown in Figure 19.

The deletion and insertion of prefixes could also result in fragmentation in the pages. To overcome this problem, all the pages that have the same page ID have prefixes sorted in the increasing order of prefix length. We then adopt the technique discussed in [20] to efficiently insert and remove entries within the TCAM to reduce fragmentation.

H. Results and Case Studies

In this section we present the results based on the architecture proposed in the paper. The results have been illustrated with two real-life large routers (`bbnplanet` and `attcanada`) to demonstrate the effectiveness of our approach. Using CACTI-3.0 model we first show that reducing the bit length and number of entries in the TCAM reduces power and access time respectively. Based on the power and memory model discussed in Section III.D we compute the value of w_1 for which power is bounded and memory requirement is least. Finally, we evaluate the results of our approach for above two routers to show their performance.

1. Power and Delay Analysis

The proposed approach is based on variation of number of bits in the 2nd level and compacting the routing table based on prefix properties. The simulation results show that these two factors affect the power and delay in the lookup process. We have used the modified version of CACTI-3.0 [22], an integrated cache area, time, and power model, to analyze the delay and power of a TCAM. The CACTI-3.0 model was developed for analysis of cache memories, so we modified the source code of a fully associative cache to model the behavior of CAM. This approach is also evident from a prior study [21] as CAM operation is equivalent to tag matching in a fully associative cache. We use 0.18um process technology parameters to evaluate the power consumption and lookup time in CAM.

2. Trends in Access Time

Using the modified tool, we found that there is a no significant change in delay with change in the number of tag bits. This is because the number of tag bits (column lines) is negligible compared to the number of row lines (prefix entries). However, changing the number of prefixes by compaction has a significant impact on the access time. Also, our technique uses small partitions for prefix comparisons, which will further reduce the number of rows being activated during an access. The access time of TCAM model, plotted in Figure 20 for 24 tag bits, bolsters the argument for compacting the routing table to reduce the access time.

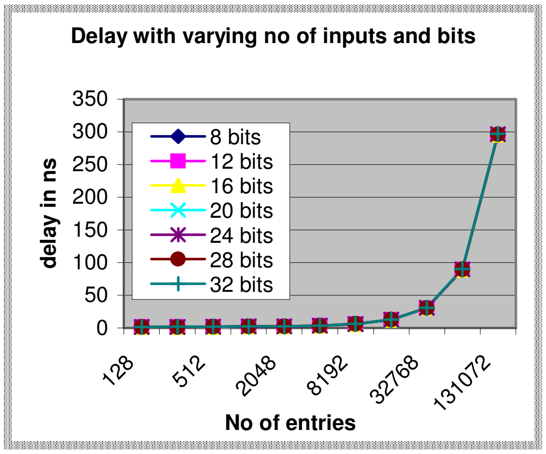


Figure 20: TCAM Entries vs Access Time

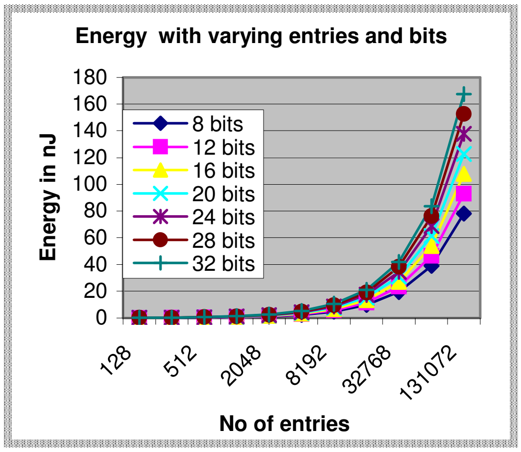


Figure 21: Number of Entries vs Energy Consumption

3. Trends in Energy Consumption

In case of power analysis, we found that there is a noticeable change in the energy consumption with the change in number of tag bits. There is an average improvement of 17.38%, when we change the number of tag bits from 32 to 24. If we further reduce the tag size from 32 bits to 16 bits, then we get 35.59% energy savings on an average. We provide the plotted results of change in energy consumption with change in number of entries and tag size in Figure 21.

4. Power Consumed per Lookup

Section III.D gives an empirical power model in terms of active TCAM entries for computing the maximum power consumption during lookup for different values of w_1. Based on the equation we find the range of values of w_1 for which power is bounded. We know that the power consumption is based on number of entries looked up in comparators, page table, pages and the bucket. As discussed earlier, the number of entries in the comparator and page table is constant and negligible for a particular value of w_1. However, the number of active pages and buckets looked up depends on the match/mismatch in the 1st level TCAM. Thus, the bound on maximum power consumption per lookup depends on the size of the bucket and the number of active pages looked up in the 2nd level TCAM. From Lemma III.1 we know that the number of entries looked up in the pages is a constant. In order, to find all the values of w_1 for which power is bounded we examine all values of w_1 for which the bucket size is $\leq$ page size. For these values of w_1 we can conclude that power is bounded by the number of pages enabled in 2nd level (256*3). We have used $\alpha_{f,}$ fill-factor for bucket as 0.5 in our simulation, which ensures that 50% of the bucket is available for future updates. We have also used a fill factor (γ) of 0.5 in the page filling heuristics.

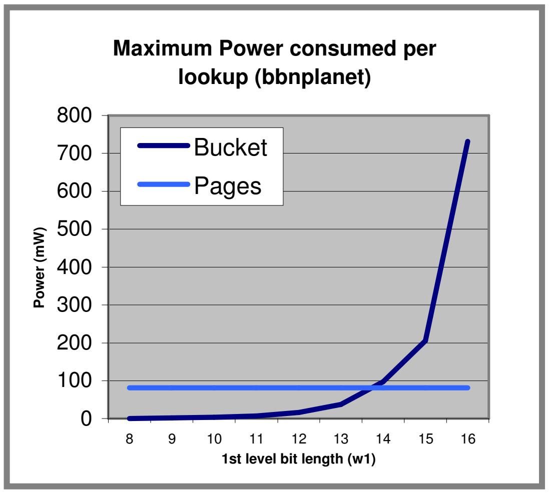


Figure 22 (a): Power Consumption in `bbnplanet` Router

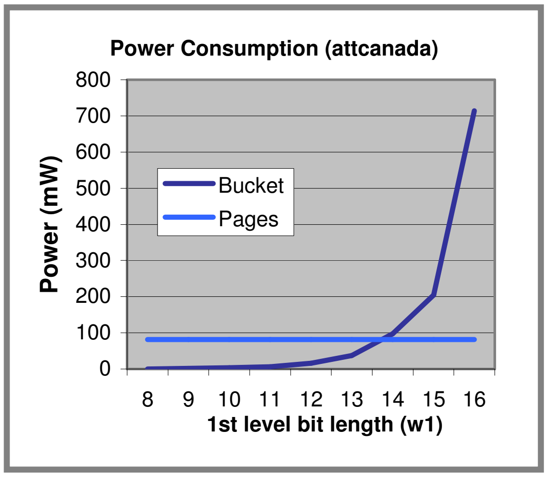


Figure 22 (b): Power Consumption in `attcanada` Router

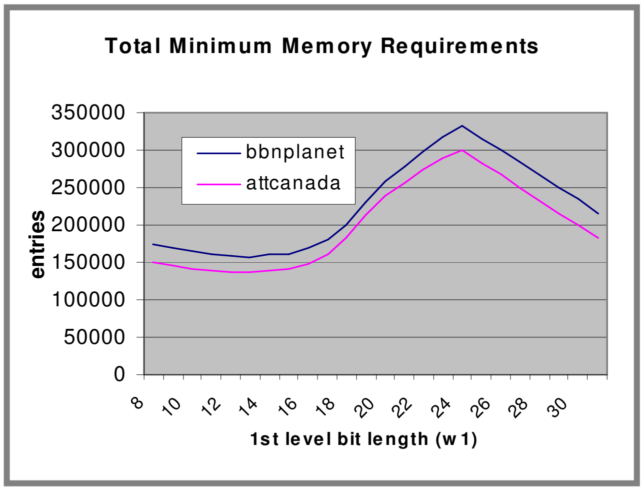


Figure 23: Total Memory Requirements

Figure in 22a and 22b show the results for `bbnplanet` and `attcanada` routers respectively. We plot the number of active entries in the bucket during lookup operation for varying w_1 and keeping the page entries to its bound. Using modified CACTI-3.0

model, the power consumption has been estimated (assuming the TCAM to run at 100 MHz). It is evident from the figures that for $w_1 \leq 13$, we obtain a bound on the power consumption for both routers.

It may be worthwhile to mention that we have not shown the values for $w_1 > 16$, since for such values of w_1 the power consumption per lookup was very high and not useful to router design.

Table 9: Reduction in Memory Requirements

Router	Raw data (entries)	After Compaction (entries)	Effect of Architecture (entries)
Attcanada	112412	57837	50182
Bbnplanet	124538	71500	59883

5. Memory Requirements

The total memory requirements for different values of w_1 are given by the empirical equation in Section III.D. We use this equation to find the optimal value of w_1 for which power is bounded and memory requirement is the least. From the above discussion we observe that for `bbnplanet` and `attcanada`, power is bounded for $8 \leq w_1 \leq 13$ (Figure 11a and 11b). We use this information to find the optimal value of w_1 for which memory is the least. From the plot given in Figure 23, we see that the value of w_1 for which memory is least for the bounded range of power is 13 for `bbnplanet` and 12 for `attcanada`.

It is important to mention that we have not considered the value of $w_1 < 8$, throughout our discussion in the paper, since we observed an increase in memory consumption when w_1 is decreased from 8 through 1, though the power requirement is bounded by the 256*3 page entries.

6. Case Studies

Based on the results from simulation and CACTI-3.0[2] model we illustrate the benefits of using the EaseCAM architecture for router design. Table 9 shows the reduced memory

[2] The CACTI model for CAM was designed by Dr.Bhuyan and Banit Agarwal of UC Riverside.

requirement for both the routers as we apply compaction and architectural technique successively.

We see that about 40% compaction takes place in the `bbnplanet` router with access time reduced by about 50%. Also, a compaction of 45% in the `attcanada` router reduces the access time by about 65% as shown in Table 10. However, based on the proposed architecture the access time is bounded by the number of active pages entries and found to be less than 5ns for both the routers.

Table 10: Reduction in Access Time

Router	Raw data (ns)	After Compaction (ns)	Effect of Architecture (ns)
Attcanada	240.53	81.87	4.39
Bbnplanet	265.32	117.9	4.43

Table 11 shows the power reduction due to the compaction and subsequently applying the architectural techniques. The power estimation was done assuming the TCAMs to run at 100MHz. The large routers like attcanada and bbnplanet can be designed with power as low as 0.135 mW per lookup using the EaseCAM approach. This is less than 1% of the total TCAM power requirement.

Table 11: Reduction in Power

Router	Raw data (W)	After Compaction (W)	Effect of Architecture (W)
Attcanada	14.35W	7.38W	0.135W
Bbnplanet	15.9W	12.31W	0.12W

IV. CONCLUSION

The main contribution of my thesis consists of a scalable and efficient algorithm for fast lookup based on modified LC-Trie technique and a storage and power efficient router architecture using TCAM. The modified LC-Trie technique performs about four times better in terms of access time in the average case as compared to the LC-Trie approach. The estimated storage requirement also fall by factor 2.38. This proposed algorithm is also efficient in terms of power utilization when compared to the LC-Trie approach. It has been predicted that the modified LC-Trie approach is easily scalable and can meet the traffic demands for atleast the next three years.

The proposed algorithm does approximately 6.6 million lookups per second on a 32 bit, 200Mhtz processor. This however does not consider caching of packets into consideration. Since the packet have certain amount of locality in them, caching could lead to better performance. Since only a few prefixes need to be searched and the next hop for the rest of the packets can be obtained from the cache. An analysis of the proposed algorithm with caching could be done to compute the maximum throughput of the proposed algorithm.

I have also presented a novel architecture for a TCAM-based forwarding engine. The results show significant reduction in memory consumption based on the prefix compaction and architectural design. Heuristic have been designed to store entries in TCAM pages so that only a bounded number of entries are looked up during the search operation. A fast incremental update algorithm has been introduced that is time bounded. The proposed scheme also tackles the memory management problem efficiently. The memory requirements, power consumption and delays for router architecture have been outlined. To demonstrate the merit of the proposed architecture, the architectural features on `bbnplanet` and `attcanada` routers based on their trace statistics have been used. It has been shown that the memory requirement is reasonably low due to use of effective compaction technique. At the same time, the power consumption is found to be remarkably low to promise efficient TCAM design in the future.

REFERENCES

[1] P. Gupta, “Algorithms for routing lookups and packet classification,” Ph.D. thesis, Stanford University, Dec 2000,[Online].
Available: http://klamath.stanford.edu/~pankaj/thesis/thesis_2sided.pdf

[2] K. Sklower, “A Tree Based Routing Table for Berkeley Unix,” in *USENIX Winter Conference*, 1991, pp. 93-104.

[3] M. Waldvogel, G. Varghese and J. Turner and B. Plattner, “Scalable High Speed {IP} Routing Lookups,” in *Proc. ACM SIGCOMM '97*, 1997, pp. 25-36.

[4] T. Chiueh and P. Pradhan, “Suez: A Cluster-Based Scalable Real-Time Packet Router,” in *International Conference on Distributed Computing Systems*, 2000, pp. 136-143.

[5] T. Chiueh and P. Pradhan, "Cache Memory Design for Network Processors," in *IEEE HPCA*, 2000, pp. 409-418.

[6] T. Chiueh and P. Pradhan, “High-Performance IP Routing Table Lookup Using CPU Caching,” in *IEEE INFOCOM ‘99*, 1999, pp. 1421-1428.

[7] E. Fredkin, “Trie Memory,” *Commun. of the ACM*, 1960, pp. 490-500.

[8] W. Szpankowski, “Patricia Tries again Revisited,” *J. ACM*, vol. 37, pp. 691-711, Oct 1990.

[9] P. Newman, G. Minshall, T. Lyon, and L. Huston, “IP Switching and Gigabit Routers,” *IEEE Commun. Magazine*, 1997, vol. 35, pp. 64-69.

[10] S. Nilsson and G. Karlsson, “Fast Address Look-Up for Internet Routers,” in *Proc. IEEE Broadband Commun. ‘98*, 1998, pp. 9-18.

[11] V. Srinivasan and G. Varghese, “Faster {IP} Lookups Using Controlled Prefix Expansion,” *Measurement and Modeling of Computer Systems*, 1998, vol. 17, pp. 1-10.

[12] G. Chandranmenon and G. Varghese, “Trading Packet Header for Packet Processing,” in *Proc. ACM SIGCOMM 95*, 1995, vol. 25, pp. 1-10.

[13] “Intenet Performance Measurement Analysis Project,” University of Michigan and Merit Network, Mar. 2001. [Online]. Available: http://www.merit.

[14] F. Zane, G. Narlikar, and Anindya Basu, “CoolCAMs: Power-Efficient TCAMs for Forwarding Engines,” in *IEEE INFOCOM ‘03*, 2003, pp. 42-52.

[15] R. Panigraphy and S. Sharma, “Reducing TCAM Power Consumption and Increasing Throughput,” in *Proc. Hot Interconnects X*, 2002, pp. 107-115.

[16] P. Shivakumar and N. Jouppi**, “**Cacti 3.0: An Integrated Cache Timing, Power and Area Model,” *Western Research Lab (WRL) Research Report*, Compaq, 2001.

[17] H. Liu, "Routing Table Compaction in Ternary-CAM," in *Proc. IEEE Micro,* 2002, pp. 58-64.

[18] R. Brayton, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.

[19] V. Srinivasan and G. Varghese, “Fast Address Lookups Using Controlled Prefix Expansion,” *ACM Trans. Computer Systems*, 1999, vol. 17, pp. 1-40.

[20] D. Shah and P. Gupta, “Fast Updates on Ternary-CAMs for Packet Lookups and Classification,” in *Proc. Hot Interconnects VIII*, 2000, pp. 45-56 also in *Proc. IEEE Micro,* 2001, vol. 21, pp. 177-186.

[21] M. Zhang and K. Asanovic**, “**Highly-associative Caches for Low-power Processors in Kool Chips Workshop,” in *Proc. MICRO-33*, 2000, pp. 91-98.

[22] J. Brelet and B. New, "Designing Flexible Fast CAMs with Virtex Family FPGAs," Xilinx Inc., 1999, [Online]. Available: http://www.xilinx.com/xapp/xapp203.pdf

[23] J. Wade and C. Sodini, “Dynamic Cross-Coupled Bitline Content Addressable Memory Cell for High Density Arrays,” *IEEE J. Solid-State Circuits*, 1987, vol. 22, pp. 119–121.

VITA

Name:	Ravikumar V. Chakaravarthy
Educational Background:	B.E., Computer Science, Mysore University, 2000.
Address:	297, 3rd Stage, 4th block, 5th Main,
	BEML Layout, Basaweshwaranagar
	Bangalore, India – 560079