Spring 2000

An algorithm for fast route lookup and update

Pinar Altin Yilmaz
New Jersey Institute of Technology

ABSTRACT

AN ALGORITHM FOR FAST IP ROUTE LOOKUP AND UPDATE

by
Pınar Altın Yılmaz

Increase in routing table sizes, number of updates, traffic, speed of links and migration to IPv6 have made IP address lookup, based on longest prefix matching, a major bottleneck for high performance routers. Several schemes are evaluated and compared based on complexity analysis and simulation results. A trie based scheme, called Linked List Cascade Addressable Trie (LLCAT) is presented. The strength of LLCAT comes from the fact that it is easy to be implemented in hardware, and also routing table update operations are performed incrementally requiring very few memory operations guaranteed for worst case to satisfy requirements of dynamic routing tables in high speed routers. Application of compression schemes to this algorithm is also considered to improve memory consumption and search time. The algorithm is implemented in C language and simulation results with real-life data is presented along with detailed description of the algorithm.

AN ALGORITHM FOR FAST IP ROUTE LOOKUP AND UPDATE

by
Pınar Altın Yılmaz

A Master's Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

May 2000

BIOGRAPHICAL SKETCH

Author: Pınar Altın Yılmaz

Degree: Master of Science in Computer Engineering

Date: May 2000

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering
  New Jersey Institute of Technology, Newark, New Jersey, 2000

- Bachelor of Science in Computer Engineering,
  Boğaziçi University, Istanbul, Turkey, 1998

Major: Computer Engineering

ACKNOWLEDGMENT

I would like to express my gratitude for my thesis advisor and research supervisor, Dr. Necdet Uzun for the support, understanding and encouragement he has provided me with during the whole course of my studies at NJIT. I would also like to thank Dr. Şirin Tekinay and Dr. Nirwan Ansari for participating in my committee. My special thanks go to my husband, Mete Yılmaz, for the love and kindness he has shown me. I thank Peter Teklinski of NJIT for providing important data for the simulations. This work has been partially sponsored by Fujitsu Network Communications Inc.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

1.1 Objective

The number of Internet hosts shows an exponential growth [8]. As new multimedia applications are introduced, and conveniences of the Web such as electronic commerce are discovered by more and more people, the number of users and the amount of traffic are also increasing. As higher bandwidth is offered by optical networks, router performance becomes the key.

When a packet arrives at an input interface of a router, the corresponding output interface that the packet should be sent to is looked up in the forwarding table. The lookup operation is to find the longest matching prefix associated with a destination address. The whole idea of prefixes is to aggregate the addresses so that the routing table size is kept small. More customers are choosing multiple network service providers for redundancy, load sharing and flexibility, their networks are "*multi-homed*". For these reasons, aggregation of addresses is not easy and improving, and the number of globally visible networks and thus the routing table sizes are increasing. More than 25 percent of prefixes are multi-homed and non-aggregatable and showing a steep linear rate of growth [9].

With the introduction of CIDR [5], prefix lengths do not have to be 8,16,or 24 for Class A, B and C networks respectively, but they can be of any length between 1 and 32. The prefix length distribution of a routing table is not uniform, showing a high concentration at 16 and 24 bit prefixes as in Figure 1.1, hitting maximum at 24 bits with more than 26,000 entries in this table of more than 47,000 entries.

As studied in [9] and [12], degradation in Internet hierarchy and the resulting increase in the globally visible paths will result in much larger routing tables and a potentially higher routing instability, which increases linearly. Routing update information tends to be extremely bursty. Core Internet routers receive bursts of updates at rates exceeding 100 prefix announcements per second. Most Internet outages are short-lived, lasting in the order of seconds or minutes. Informal experiments with several routers suggests that sufficiently high rates of pathological updates (300 updates per second) are enough to crash a widely deployed, high-end model of Internet router [9].

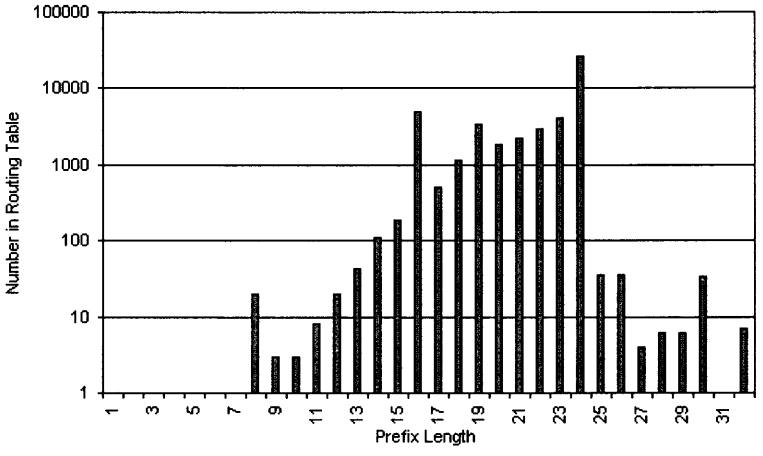

Figure 1.1 Histogram of prefixes from Mae-East NAP on September 2, 1999 (logarithmic scale) [12]

1.2 Internet Routing Instability

The Internet is divided into a large number of distinct regions of administrative control, commonly called *autonomous systems* (AS). An autonomous system (also called a routing domain) typically consists of a network service provider or a larger

organizational unit, such as a college campus or a corporate network. Each AS connects a number of subnetworks, such as remote corporate offices or customer networks. Autonomous systems usually have distinct routing policies and connect to one or more remote autonomous systems at neutral private or public *exchange points*. An *inter-domain (or exterior) routing protocol* is used to exchange information between peer routers in different autonomous systems. An *intra-domain (or interior) routing protocol*, in contrast is used to pass information between routers within an autonomous system. Internet routers build routing tables based on topological information conveyed in routing control messages exchanged with other routers.

Ten to twelve large Internet service providers dominate the Internet. These national and international providers, often referred to as *tier-one providers*, account for the majority of routes and bandwidth that compromise the public Internet. Approximately four to six thousand smaller regional networks, or *tier-two providers* peer with the tier-one providers at one or more private or public *Internet eXchange Points* (IXPs). These large exchange points are considered the core of the Internet where providers peer, or exchange both routing information and traffic.

Backbone service providers participating in the Internet core must maintain a complete map, or "default-free" routing table, of all globally visible network-layer addresses reachable throughout the Internet. At the boundary of each ISP backbone, peer border routers exchange reachability information to destination IP blocks, represented as prefixes.

Border Gateway Protocol (BGP) is the most common inter-domain (exterior) routing protocol used by autonomous systems in the Internet. BGP is an incremental

protocol using TCP. BGP information includes attributes such as address of next-hop router and a record of inter-domain path the router has followed through different providers to detect and prevent routing loops called ASPath. Routing information in BGP has two forms: announcements and withdrawals. A BGP update may contain multiple route announcements and withdrawals.

Internally, within an autonomous system, routers use a variety of intra-domain (interior) protocols to distribute local routing information, including Open Shortest Path First (OSPF), Interior Gateway Routing Protocol (IGRP), and Intermediate System-to-Intermediate System (ISIS). Exterior gateway protocols exchange routing updates between adjacent neighbors, however, an inter-domain router floods information to all other routers throughout the network.

After a policy change or network failure affects the availability of a path to a set of prefix destinations, the routers topologically closest to the failure will detect the fault, withdraw the route and make a new local decision on the preferred alternative route, if any, to the set of destinations. These routers will then propagate the new topological information to each router within the autonomous system. The network's border routers will in turn propagate the updated information to each external peer router, pending local policy decisions. Routing policies on an autonomous system's border routers may result in different update information being transmitted to each external router. The interaction between internal and external gateway protocols varies based on network topology and backbone provider policy. Most providers aggregate information at the backbone boundary.

Routing instability is the rapid change of network reachability and topology information. It may be caused by router configuration errors, transient physical and data link problems, and software bugs. High levels of network instability can lead to packet loss, increased network latency, and time to convergence. A router which fails under heavy routing instability can instigate a "route flap storm". In this mode of pathological oscillation, overloaded routers are marked as unreachable by BGP peers as they fail to maintain the required interval of Keep-Alive transmissions. Then peer routers will transmits updates reflecting topology change. When the failed router recovers will initiate a BGP peering session generating large state dump transmissions. This increased load will cause more routers to fail and a storm propagates through the Internet. Several route flap storms have caused extended outages for several million network customers. Latest generation of routers from several vendors provide a mechanism in which BGP traffic is given a higher priority and Keep-Alive messages persist even under heavy load.

Here are some of the results:

- Routing information is dominated by pathological or redundant updates which may not reflect changes in routing policy or topology.
- A strong correlation is found between network usage and inter-domain routing information at the major IXPs, contrary to the expectation of an exponential distribution for the inter-arrival time of routing updates. It also exhibits daily and weekly cyclic trends.
- Routing instability increases linearly.

- Routing update information tends to be extremely bursty. Core Internet routers receive bursts of updates at rates exceeding 100 prefix announcements per second.
- Instability and redundant updates exhibit a specific periodicity of 30 to 60 seconds.
- Instability is not dominated by a small set of autonomous systems or routes. It is well distributed over destination prefixes, peer routers, and origin autonomous systems.
- There is no correlation between the size of an AS (as measured by the number of routes it is responsible for in the routing table) and its proportion of the instability statistics.
- Most Internet outages are short-lived, lasting in the order of seconds or minutes.
- Significant levels of BGP instability stem from congestion collapse (as described above as rout flap storm).
- Most routing problems stem from human error and misconfiguration of equipment.

Degradation in Internet hierarchy and the resulting increase in the globally visible paths will result in much larger routing tables and a potentially higher routing instability.

Informal experiments with several routers suggests that sufficiently high rates of pathological updates (300 updates per second) are enough to crash a widely deployed, high-end model of Internet router. A crash is a state in which the router is completely

unresponsive and does not respond to future routing protocol messages or console interrupts.

A number of vendors have implemented router dampening algorithms to hold down, or refuse to believe, updates about routes that exceed certain parameters of instability, such as exceeding a certain number of updates in an hour. Dampening algorithms can introduce artificial connectivity problems, as legitimate announcements may be delayed. [9]

1.3 Practical Considerations

Routers can be seen in two categories:

1. Backbone Routers
    - run inter-domain (exterior) protocols between different autonomous systems.
    - have routing tables of around 45,000 or more prefixes.
    - route changes occur over 100 times per second.
    - require frequent reprogramming.
    - distribution of packet sizes is bimodal, with peaks corresponding to either 64 byte control packets or 576 byte data packets (for wide area traffic).
2. Enterprise Routers
    - run intra-domain (interior) protocols within an autonomous system.
    - have routing tables of around 1000 prefixes.
    - route changes occur once every few seconds.

- distribution of packet sizes is bimodal, with peaks corresponding to either 64 byte control packets or 1519 byte data packets (for local area traffic).

Large multi-campus enterprise routers are much more like backbone routers [16].

Forwarding in the router is accomplished by two different approaches:

1. *Wire Speed*: Router does not have large buffers to store incoming packets and forwards them immediately at the same speed as the input interface. Router must have a fast lookup algorithm so that when the last bit of the packet arrives, the routing decision is made and the packet can be forwarded to the output interface. The shortest IP packet is 64 bytes and this corresponds to 40 nsec at OC-192 speed.
2. *Probabilistic Estimate*: A probabilistic model of packet size and arrival times is assumed and router has a large buffer accordingly so that when the network behavior is different from the model, packets can be buffered and processed. Worst case behavior of the router is not as important as the average case behavior in this sense.
3. The number of distinct next-hops in the routing table of a router is limited by the number of other routers or hosts that can be reached in one hop, so it is not surprising that these numbers can be small even for large backbone routers. Therefore, any algorithm can take the advantage of keeping pointers to a next-hop table instead of next-hops themselves. However, if a router is

connected to, for instance, a large ATM network, the number of next-hops can be much higher.

4. There are several alternatives in the internal design of a high-speed router. Routing computation is based on exchanging routing protocol messages with other routers and composing a routing information base (RIB). A set of information is derived from RIB to form a forwarding information base (FIB) to be used by forwarding engine(s). A router has one or more forwarding engine to perform a longest matching prefix lookup in FIB. A router may have one forwarding engine per input interface. Forwarding engines may have a shared FIB, or its own FIB to ensure higher performance. Fast lookup algorithms can be implemented in either the routing computation module or forwarding engine. Several issues must be considered when evaluating a fast lookup algorithm:
5. *Memory Requirement:* The amount of memory consumed by an algorithm in its particular data structures is a cost factor. In an implementation requiring many FIBs, small memory consumption is desirable. The amount of memory should be considered as the total memory that will be used in all nodes in the network. It is desirable to keep the memory requirement down. If the data structure allows to keep entities naturally aligned, then expensive instructions and cumbersome bit extractions may also be avoided.
6. *Search Algorithmic Complexity:* The complexity of an algorithm may be based on several factors:
    - the maximum length of prefixes (32 bit for IPv4 or 128 bit for IPv6),

- the size of the routing table,
- distribution and locality of prefixes.

Best case and worst case behavior of an algorithm may be affected by different factors as well.

7. *Insertion and Deletion of Routes:* Routing tables in Internet are dynamic, meaning that insertions, deletions and modifications of routes are frequent. An algorithm may be able to provide incremental insertion and deletion into its particular data structure or may require that the data structure (FIB) is built periodically from RIB. Parallel execution of lookups and updates may be performed by maintaining multiple copies of forwarding table or by locking mechanisms. If there are multiple forwarding engines, instead of performing updates on each FIB, a most up-to-date FIB may be downloaded to other FIBs periodically.
8. *Handling of Exceptions*: In the real world, catastrophic situations may arise such as misconfiguration of a router or route flap storms. A graceful handling of exceptions is desirable [17].
9. *Flexibility*: The flexibility and extensibility of the algorithm to several factors is also an important issue to be considered versus the investment. Those factors include:
   - changes in existing protocols,
   - introduction of new protocols (IPv6),
   - support for additional features such as multicasting and multi-path routing,

- changes in routing table characteristics such as prefix distributions over the address space and prefix length distributions.

1.4 Thesis Organization

In Chapter 1, Internet routing instability is discussed in the light of the "Internet Performance Monitoring and Analysis (IPMA)" project [9,12]. Practical considerations in router design will also be discussed In Chapter 1. Chapter 2 will present a summary of some of the available schemes for routing lookup. Chapter 3 will introduce LLCAT algorithm and discuss its performance in the light of simulation results. Chapter 4 compares several schemes based on published data, and Chapter 5 will present the conclusions.

CHAPTER 2

CLASSIFICATION OF ROUTING LOOKUP ALGORITHMS

2.1 Trie-based solutions

2.1.1 Patricia Trie

Generic search algorithm, Practical Algorithm to Retrieve Information Coded in Alphanumeric (Patricia), is modified with an additional invariant to maintain a routing trie. Patricia trie is a binary radix trie with one-way branching removed. It is also referred to as a path compressed binary trie because it actually eliminates the traversal of the full path to a leaf node if there is only one leaf in that branch by introducing a skip factor, which is the length of the path to be overlooked. It suffers from backtracking which occurs when a match is not found at a leaf node. The search goes up to parent node by explicit parent pointers at each node. Because of the backtracking, worst-case complexity is $O(W^2)$ where W is number of bits in the address. It is noted that in the average case Patricia tries are approximately balanced. The expected length of a search is 1.44 log (N) where N is the number of entries in the routing table [15]. This is probably the most popular and most widely implemented scheme starting with the implementation in 4.3 Reno release of Berkeley UNIX, mainly due to the fact that it is easy to implement in software.

2.1.2 Dynamic Prefix Tries

To simplify deletion and avoid recursive backtracking in Patricia tries, dynamic prefix tries have been introduced. At each node, a link to its parent, left child and right child the routing decisions for left and right branches are stored with index key, which

determines how many bits are common in the prefixes inserted in the branch. The index key allows to avoid one-way branching. The look-up times on such a trie have an upper bound of 2×W iterations. Insertion and deletion operations do not depend directly on the size of the trie but linearly on the height of the trie [4]. It was observed that the average search time increases linearly as a function of log(N) which indicates that the tree is quite balanced.

2.1.3 Level-Compressed Tries

The total number of nodes in a path compressed binary trie is exactly $2n$-1, where n is the number of leaves in the trie. Path compression is a way to compress parts of a trie that are sparsely populated. Level compression is for compressing parts of a trie that are densely populated. The idea is to replace the i highest complete levels of the binary tree with a single node of degree 2^i where i is called the branching factor. The worst case look-up is bounded by the maximal path through the trie. The average depth of the trie grows very slowly, it is O(log log n) for a large class of distributions. The routing table consists of four parts, Level-Compressed-Trie (LC-trie), base vector, next-hop table, and prefix vector. Base vector is a sorted array of complete prefixes with pointers to next-hop and prefix table entries. Next-hop table contains all possible next-hop addresses, and prefix vector contains information about prefixes which are proper (fully contained) prefixes of others. The LC-trie is constructed in a top-down fashion from base vector and is implemented as an array. The branching factor at the root of the trie has a large influence, it is particularly advantageous to choose a large branching factor for the root. The authors carried on simulations on two data sets [14].

For one set they used a permutation of the routing table to generate packet traffic for which packet trace was not available. For the other, they used actual real life packet traces. With actual traffic traces, they observed a lookup time twice as fast even though the trie was larger and deeper (41,000 entries, 2.3MB) which can be explained by the locality in traffic traces. The maximum depth of the trie, was observed as 5, average is just below 2. Multicast support is available by treating multicast addresses as 32-bit prefixes using a bit flag. Multipath routing can be implemented similarly. The authors do not mention affects of updates and how they can be performed. LC-trie can be built periodically from base vector which has to be maintained sorted, or sorted periodically as well [13,14].

2.1.4 Expanded Tries

The idea is to reduce a set of arbitrary length prefixes to a predefined set of prefixes by prefix expansion. In each multi-bit node of the trie there will be a portion of the prefixes, sub-prefixes of the same length. Shorter prefixes will be expanded into many entries by padding bits. The resulting trie may have the same length of prefixes, e.g. 8-bit sub-prefixes at each level or different length of sub-prefixes at each level, e.g. 16,8 and 8 bits at three levels. Such tries are *fixed-stride*. In each level of the trie, there may be nodes carrying prefixes of different length, e.g. at the same level, say 2, there may be a node carrying 8-bit prefixes and another carrying 4-bit prefixes, such tries are *variable-stride*. To improve locality and storage requirements, leaf-pushing and packed arrays can be used. When the prefixes are expanded, the prefix length information which is necessary for updates is lost. The solution is to keep the original

prefix table intact, update this table and generate the trie from scratch when there are changes in the table as described by the authors in US Patent #6,011,795. The recomputation of the trie may be expensive when a new prefix is inserted especially in the longest stride. Another solution suggested by the authors is to maintain original unexpanded prefix information separately. One-bit trie for each node is used to store actual prefix data before expansion. Fixed strides are desirable because of their simplicity, fast optimization times, and faster search times. Search time is linear on the number of levels of the trie. Updates depend linearly on length of the prefix and maximum size of the trie node[16].

2.1.5 Multiway and Multicolumn Search / 6-way Search

In this scheme, each prefix is treated as a range and encoded using start and end of range. Thus, the longest match lookup has been translated to the problem of finding the narrowest enclosing range where any region in the binary search between two consecutive numbers corresponds to a unique prefix. Entries are arranged in a binary search table and a mapping between consecutive regions in the binary search table and corresponding prefixes is precomputed. The approach is to do a binary search on the number of possible prefixes. An initial array for the first 16-bit prefixes can be used as a front end to reduce the number of keys to be searched in binary search. The initial array replaces a single binary tree by several smaller binary trees. Without the initial array, worst-case possible number of memory accesses is $log_2 (N+1)$ which can be 16 or more lookups for large tables with N prefixes. With the initial array, worst case lookup takes 10 memory accesses for 38,000 entries. Locality inherent in processor

cache is exploited and a multiway search is possible. For 38,000 entries, table building time was 5.8 sec and memory consumption is 950KB, for 700 entries 350 msec and 265 KB respectively. The authors estimate a worst case insertion time of 300 msec for a table of 38,000 entries when implemented in software. However, it is also noted that building a table from scratch achieves better results in the order of *O(N)* where there are *N* prefixes in the table[10].

2.1.6 Lulea Scheme

This scheme uses a trie with fixed levels of 16,8, and 8 bits. The result of a search on a level is either an index into next-hop table or an index into an array of chunks for the next level. It optimizes the information associated with prefixes by noting that there are typically only a small number of next hops. Large trie nodes can be compressed using a method of counting the number of bits set in a large bitmap, which results in 150-160KB memory requirement and 99 msec to be built (Pentium Pro 200Mhz 256 KB L2 cache) for a table with 40,000 prefixes. Leaf pushing is used to complete the trie at each level, thus incremental insertion and deletion will be slow, and forwarding table is built during a single pass over all routing entities. Table size and building time is linear in the number of routing table entries. In the compression scheme, there are assumptions for the routing table characteristics which may not hold in the future, and the compression scheme will have to be modified. For IPv6, the data structure can be tuned to the properties of the routing table when it is available[3].

2.1.7 Multi-Resolution Tries

It is actually an expanded trie with fixed strides. Each node of the trie is an m-structure. M-structures contain forwarding information in p-structures that are entries of the node. The rest of the m-structure is the necessary information for updates where actual prefix information (length and decision) is stored as a linked list per entry. These linked lists can be sorted or organized into a trie for faster access. The idea is p-structures can be separated from the node and stored in FIBs using less storage. Garbage collection is necessary to remove nodes with no decisions and no children. The authors have implemented the trie with a small strides with 12,4,4,2,2,2,2,1,1,1,1 bits, which generates a worst case lookup of 11 memory accesses, on average 4 memory accesses. A routing table of 40,000 entries fit into 1 MB of memory. The ideas of using variable strides and leaf pushing also apply to this scheme to decrease storage requirement, but they have a negative impact on update operations [17].

2.2 Hashing Based Solutions

2.2.1 Caching

Instead of pushing the performance of packet processing, an alternative approach is to avoid repeated computation by applying the idea of caching to network processing. Because the data streams presented to Internet processors and general purpose CPUs exhibit different characteristics, cache design trade-offs for the two also differ. Caching alone is not sufficient due to less locality in packet address streams than the instruction/data reference streams in program execution. Internet addresses in lookup requests exhibit temporal locality as opposed to spatial locality exhibited by program

references. To improve the cache performance, effective coverage of the IP address space is achieved by address merging. In this scheme, the destination host address is treated as a memory address, and each cache entry corresponds to a host address range. The distinct outcomes of routing table look-up is relatively small and equal to the number of output interfaces. Therefore, a hash function can be used to combine disjoint host address ranges that share the same routing table look-up result into a larger logical address set. Average routing table look-up time depends on both cache hit ratio as well as cache miss penalty, which is determined by the look-up algorithm. It is concluded that the block size of a network processor cache should be small, preferably one entry wide, because network packet streams lack spatial locality, which makes it infeasible to depend entirely on caching. There seems to be sufficient temporal locality to justify the use of cache in Internet processors. This scheme achieves less than 2 percent miss ratio in simulations using real packet data from Brookhaven National Laboratory with cache size being 4KB or 8KB. State-of-the-art designs for caching does help to improve performance of IP lookups, however, it does not eliminate the need for fast address lookups algorithms [2].

2.2.2 Content Addressable Memories (CAMs)

CAMs can be used to implement exact matching by using a separate CAM for each possible prefix length, or CAMs that allow "don't care" bits can be used so that a single CAM design is possible. Largest CAMs today allow around 8000 prefixes, which will be insufficient for backbone routers. CAM designs did not keep pace with the improvements in RAM memory, a CAM solution runs the risk of being made

obsolete in a few years by faster processors and memory [11]. A new form of CAMs called ternary CAMs have been proposed. However, they suffer from high cost, large power dissipation, and O(N) worst case update times.

2.2.3 Large Memory Architecture

Assuming memory is cheap, a fast solution based on pipelining can be deployed that consumes a significant amount of memory is introduced. This scheme keeps two tables in DRAM. The first 24 bits of the address is used as an index into the first table. Since there are few prefixes longer than 24 bits, most look-ups take one memory access time. If the prefix is longer, a second look-up is performed. This is a two level pipelining. To refine the scheme, multiple levels, thus multiple tables can be used, which increases the average number of memory accesses per look-up, but decreases memory requirements. However, in this scheme, updates are not trivial and may require that a large number of memory accesses be made. The authors suggest that dual memory banks shall be used to facilitate the updates, which doubles already excessive memory requirements. If a single memory bank is used then the update instructions to be sent to the hardware from the microprocessor becomes a burden. With enhancements, the update message can be one instruction, however, still many entries have to be rewritten, and hardware update mechanism have to be designed carefully [7].

2.3 Hybrid Solutions (Trie and Hashing)

2.3.1 Binary Search on Levels

Evolving the idea of maintaining a hash-per-network mask length used before advent of CIDR, an algorithm utilizing *binary search on levels*, on the prefix lengths is proposed. Hashing is used whether an address matches any prefix of a particular length. Binary search is deployed to reduce number of searches from linear to logarithmic. To prevent backtracking, precomputation is used and markers are placed in the hash table. Each hash table (markers plus prefixes) can be thought of as a horizontal layer of a trie corresponding to some length L except that the hash table contains the complete path to that layer of each entry in that layer. At the start of search, a hash on prefixes provides the median of the trie, if it matches, the upper half of the trie is searched, lower part otherwise. Precomputed markers of different complexities in the trie ensure that the search is progressing within the correct sub-range of the address space. Memory usage was less than 1.4MB for a table of 33,000 entries [19]. Worst-case time complexity is encountered if the organization of prefix lengths is balanced. It is non-trivial to incrementally insert or delete entries, so practically the table has to be rebuilt periodically to ensure the expected search times. Since hashing tends to waste some memory or can generate excessive collisions if non-perfect hashing functions are employed, their choice can have a significant impact on performance. The real advantage is the potential scalability it offers for IPv6.

2.4 Protocol based solutions

IP and Tag switching rely on the idea that best matching prefix can be replaced by an exact match by having a previous hop router pass an index into the next router's forwarding table. The cost is additional protocol and potential set up delays. IP switching depends on long-lived flows, and may be ineffective for short-lived flows such as web sessions. Both schemes require large parts of the network to make the required protocol changes. Adding a new protocol that interacts with every other routing protocol may increase the vulnerability of an already fragile set of protocols. Neither schemes can completely avoid ordinary IP look-ups [16,19].

CHAPTER 3

LLCAT: LINKED LIST CASCADE ADDRESSABLE TRIE

3.1 Basic Idea

LLCAT is an expanded trie with fixed strides, which makes memory management very simple. The basic idea of expanded tries has been described in [16]. However LLCAT algorithm differs from expanded tries presented in the literature in the fact that it has a new approach for memory management and update operations where it is necessary to maintain original prefix data which is lost during expansion. The idea is to represent the trie in a way most suitable for hardware implementation. The trie nodes are represented as segments in memory. For IPv4, the first stride is 8 bits since there are no prefixes less than 8 bits. The rest of the address is divided into blocks of K bits. K may be 4 or 8. For IPv6, a larger K may be appropriate. The search is bounded by the number of levels in the trie. If K=8, then it takes 4 memory accesses in the worst case. The improvement of LLCAT over other expanded tries is that it provides a more efficient approach for updates. Instead of keeping tries or linked lists to hold actual prefix information that is lost in the trie node due to expansion, it keeps a bitmap of inserted prefixes in each entry, or word in memory, and it distributes the actual prefix decisions over entries in the trie node. Update operations depend on number of levels, size of the trie node, and distribution of the prefixes, not necessarily on the size of the routing table [1].

We will provide a brief description of the algorithm in the following section. For a more thorough description and understanding of the algorithm, please refer to

the Appendices section, where we explain the idea and development of the data structures and present flowcharts for the procedures.

3.2 Algorithm Description

The algorithm described here is designed to be easily realized in hardware. It does not contain any complex computational functions, only data transfer and bit-wise logical operations. Even though this work is oriented toward IP routing, the algorithm may be used for other purposes, such as label-to-flow mapping, address filtering, and so forth. The algorithm uses the concepts of segment and offset. Physical memory is divided into segments, which can be uniquely addressed. Each of those segments has an identical number of words, and each word within any given segment can be identified by an offset.

The search string, destination address (DA) is divided into K equal substrings. Each of them will become an offset during the search. Every node will contain 2^K words in order to accommodate every possible combination of a K-bit substring as in Figure 3.1. Each word contains the following fields:

- *Stop Bit (S):* indicates whether the search stops here or will follow a link associated with the forward pointer (FP).
- *Life Children Counter (CC):* shows the number of words in the node that carry useful information. For an entry to be valid, there is either routing information present (VSP≠"0") and/or a forward pointer exists (S="0"). The first entry of a segment contains CC.

- *Output Port Address or Route (R):* contains an information element, which the algorithm looks for, i.e., an output port address to which the packet needs to be forwarded.
- *Valid Sub-prefix Pattern (VSP):* is a K-bit flag that contains the mask of possible prefix patterns for this word. If VSP is not "0," then a valid routing decision is present in R field. For example, if K^{th} bit has been set, then a full prefix has been inserted in this word.
- *Forward Pointer (FP):* contains the address of another segment in memory that the search will look at next if S is not "1."

The address of the first node, which is the root of the trie, is always fixed at 0. By concatenating 0 with the first substring of DA, the address of the first visited entry in memory is obtained. The address of the node of the next level is stored in the FP field, and the address of a single entry can be obtained by concatenating the content of FP from the previous level with a second substring. Again, the FP will be obtained by reading this memory location, and by concatenating it with a third substring to obtain the address of the word, belonging to the node of the second level. When the search reads the word in memory, it checks all the flags. If VSP is not "0," then R will be stored in a temporary register, so that at any point, the last valid R is available. The process continues until the S = "1" is hit. At this point, the search knows that it has to stop, and the R field at this location is used to forward this packet if VSP≠"0." If VS ="0," the last valid R stored in the temporary register is used as the forwarding address.

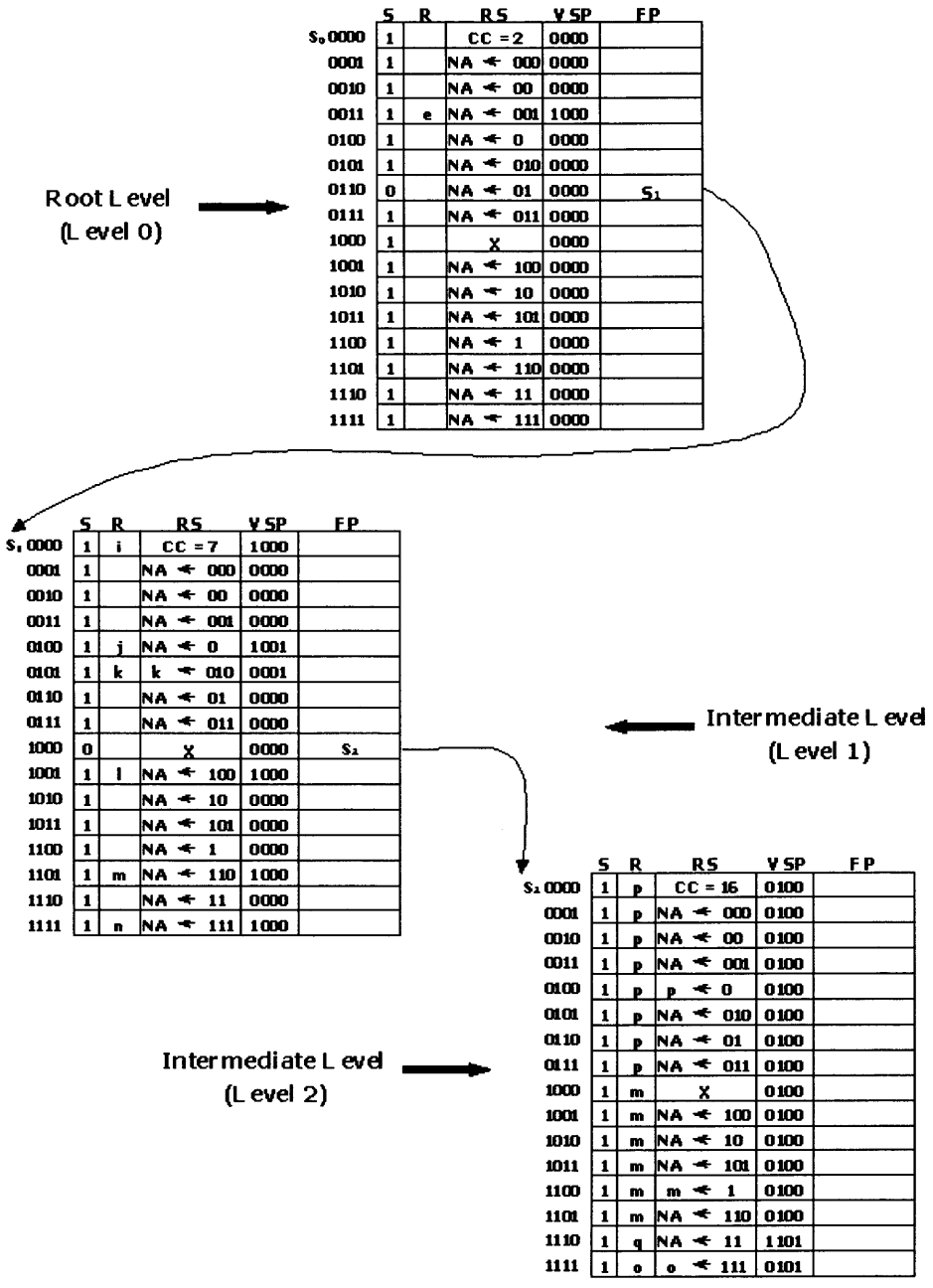

Figure 3.1 Nodes at different levels of trie

The first necessary observation to make about sub-prefixes is that their number is always two less than the number of offsets (or words) in the node, 2^K. In the case where $K = 4$, the number of words in the node is 16, and there are the following possible sub-prefixes: "0," "1," "00," "01," "10," "11," "000," "001," "010," "011,"

"100," "101," "110," and "111." There are only 14 of them, which allows them to be distributed among 16 words of the node. It is desirable to distribute the prefixes in some orderly fashion and, if possible, to have a high correlation with full prefixes. To determine the offset of a sub-prefix, the following rule applies: 3-bit sub-prefix & "1", 2-bit sub-prefix & "10," 1-bit sub-prefix & "100," where "&" stands for concatenation, constituting the offset of the word in the node where the route for the sub-prefix is to be found. As a general rule, the sub-prefix is concatenated with "1" if it is *K*-1 bit long, with "10" if it is *K*-2 bit long, and with "1" and n-1 zeros, i.e. "10..0", if it is *K*-n bit long. The unused locations of the CC field can be used for this purpose. This field is to be renamed as the route for sub-prefix (RS). Now a sub-prefix is inserted into the appropriate RS field of the last node encountered by the insertion operation in which the number of bits in the prefix is not a multiple of *K*.

Insertion of the sub-prefix is carried out by traversing down the trie and creating new nodes along the way if S="1" is encountered, until the last node is reached. For sub-prefix insertion, first, the offset of the entry is determined by the rule given above, by concatenating appropriate bits to the sub-prefix. Routing information is written to the RS field of the determined location. Then, depending on the length of the sub-prefix and its value, the insertion procedure affects several other entries of this node. Namely, it is necessary to expand the sub-prefix into all possible full prefixes. For example sub-prefix "01" is expanded into "0100," "0101," "0110," and "0111." Notice that after expansion, offsets are sequential. In terms of hardware, it may be implemented simply by concatenating values of the counter to the sub-prefix. If VSP≠"0" and S="1" for a word, then this word does not contain any information and

upon insertion, CC of the node must be incremented. Otherwise, the word has been accounted for previously and there is no need to change CC. The insertion procedure figures out which fields have to be modified. Because the sub-prefix has a length of 2 bits, the second bit in VSP fields is affected. VSP in every affected word will be OR'ed with "0010" (second bit asserted). Then the insertion procedure has to decide if it needs to replace the R field. If a decision for a longer prefix has been inserted previously, that is, if any higher order bits in VSP has been set, R field remains as it is. In other words, insertion of the sub-prefix only makes changes to routing information if it replaces routing information of a sub-prefix of shorter or equal length. Routing information of the full prefix, or longer sub-prefixes, are not replaced.

Deletion of a sub-prefix works in the following way. The sub-prefix is expanded to its full prefixes. Then, all of the entries addressed by these prefixes are examined. First, the VSP pattern is going to be changed. Corresponding bit, depending on the length of the sub-prefix, is deasserted by performing a logical AND operation with one's complement of the binary number, which has all bits "0," except for the one corresponding to the length. To clarify, if we try to remove the sub-prefix "10," then all the VSP fields associated with the full prefixes obtained by expanding "10" are AND'ed with one's complement of "0010," which is "1101." In simple terms, the deletion operation makes the second bit of every VSP "0." If any of the higher-order bits, corresponding to the longer prefixes, are asserted, the deletion procedure moves on to the next word. With any of the lower-order bits, it has to determine the longest sub-prefix asserted, and look up RS field of the corresponding entry, and insert the value of RS into R field of the current word. If no bits of the VSP are "1," then the

routing information in R has been invalidated by the deletion procedure. If S="1" then this word does not contain any more information, the word has been deleted and CC of the node must be decremented. After that, the deletion moves on to the next word.

To keep track of which segments are used in the trie, a list of idle nodes (ILL) is maintained. Each node in ILL has its FP set to the address of next node in the list. Pointers to head and tail of the list are stored separately. Nodes to be inserted are extracted from the head, and deleted nodes are added to the tail.

Observing that there are no prefixes shorter than 8 bits in current routing tables for IPv4, it is beneficial to combine levels 0 and 1 and just keep it in the small fast cache. This enhancement saves one clock cycle. Thus, the root node will have 2^8=256 words, and the first stride is 8 bits.

3.2.1 Multicast Enhancement

Multicast has become a significant part of the IP traffic. It is expected that in the future, the amount of multicast traffic will continue to grow for IPv4. IPv6 has a wide range of multicast addresses for different purposes. Therefore, the issue of speedy multicast forwarding should be addressed.

In IPv4, multicast traffic has the designated address class D, where the first four bits are "1110." In IPv6, for a packet to be multicast, it has to have "FF" in hex or "11111111" as a first byte of the DA. The value of K chosen to be 4 for IPv4 and 8 for IPv6 allows for the following: one of the FPs of the root level points to the multicast trie. The algorithm follows the link to the multicast trie, and somewhat different procedures are executed.

The difference is the following. Instead of storing values for R and RS, the pointers to multicast pattern values are stored in the combined R & RS field. This field should be sufficient to address large enough numbers of the multicast pattern. Refer to Figure 3.2. The width of the unicast word is 40 bits. If we consider the 64x64 switch, the multicast pattern occupies two memory words. Multicast patterns are located in a different part of memory. When the multicast entry is to be inserted in the trie, the unicast procedure is executed, except that when the R or RS field needs to be read or written to, the R & RS field points to the place where the actual values are stored. In the case of Figure 3.2, there are two multicast nodes shown. The node with the segment X0011 is a multicast node; if the search needs its R value, then it uses the R & RS field as a pointer to the memory location that keeps actual values of both R and RS, and it follows this link. In the case of R, the search reads as many words as a multicast pattern takes, starting with offset 0. In the case of RS, the search starts from the middle of the segment holding multicast patterns for R and RS, because it is aware of the multicast pattern size. In this case, it starts reading with offset "10."

Memory management will also be complicated by this enhancement. Instead of one ILL, there will be two: one holding nodes and another holding multicast patterns. The problem with this scheme is that while the nodes' ILL may run out very fast, the multicast patterns' ILL can still have a lot of memory, and there will be no way to transfer this resource. The problem can be resolved by more complex schemes of memory management. In Figure 3.2, only one ILL can be used, because the nodes themselves and the multicast patterns occupy four words. However, this is coincidental, and two ILLs will have to be used.

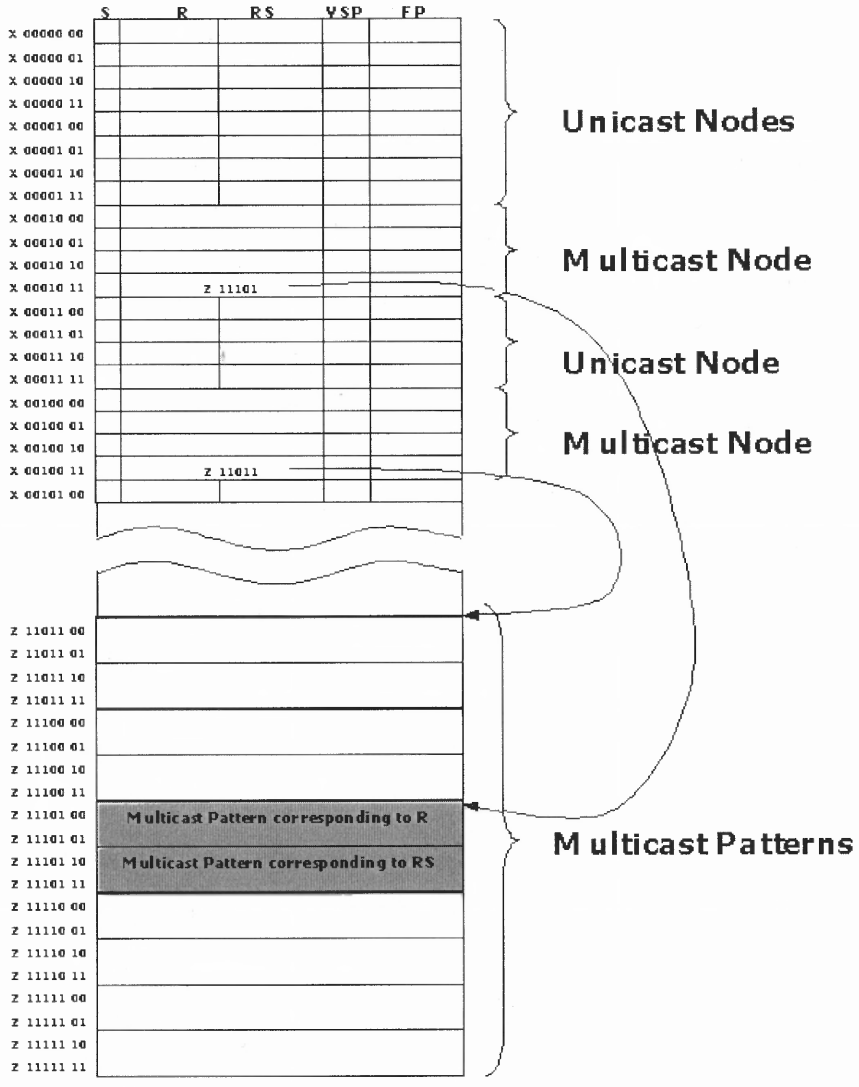

Figure 3.2 Representation of multicast node

3.3 Performance Analysis

As mentioned before in the previous subsections, the appropriate values of K for IPv4 and IPv6 are 4 and 8, respectively. In this section, the worst- and the best-case timing and memory requirements are presented. The worst-case requirements for timing are defined as the time necessary for the operation with a 29-bit prefix for IPv4 and a 121-bit prefix for IPv6. These require the longest time. The worst case for memory

requirements occurs when all the entries are located in the individual nodes of the trie. At some point, nodes each have only one valid child organizing a number of linked lists.

Number of memory operations required for search is bounded by the depth of the trie, which are 7 for K=4, and 4 for K=8 considering IPv4.

For insertion, worst case analysis is given below for K=4 for a prefix with length 29 bits:

1. Traverse root node. (1 read)
2. Observe that stop bit is 1, a new node has to be created. Get a node from ILL, and assign to forward pointer. (1 read, 1 write) There is no need to take care of CC of the root node because it is never deleted.
3. For each level traversed to last level, get node from ILL. Update the relevant entry and CC of the node.(1 read, 2 write, 5 new nodes are created.)
4. At the last level, update 8 entries covered by the prefix and update CC of node. No entries are read because the node is new. (9 write)

Totally, that would give: 1+2+3x5+9 = 27 memory operations.

For deletion, worst case analysis is given below for K=4 for a prefix with length 29 bits:

1. Traverse to node at last level, last node. (6 read)
2. Update 8 entries covered by the prefix in last node. (8 read, 8 write)
3. Update CC of last node, observe that the node will be deleted (1 read, 1 write)
4. Add last node to ILL (1 write)

5. Add the nodes from the root to the last node to ILL. (1 write for the entry in path, no read because it was in the stack, 1 read and 1 write for CC of the node, 1 write to add node to ILL, 5 nodes are deleted.)

6. Update entry in the root node. (1 write, no read because it was in the stack, no need to track CC of root)

Totally, that would give: 6+8+8+1+1+1+4x5+1 = 46 memory operations.

Calculations for IPv6 are similar.

3.4 Implementation in Hardware

For speed purposes, it is recommended that this algorithm be implemented in hardware. The circuit is responsible for three main operations: search, insertion, and removal of the entries. Whereas for insertion and removal of the entries, it is necessary to interface the microprocessor, for the search, it is not necessary. The circuit should be informed of the arrival of the new packet with an appropriate signal; at the same time, necessary fields should be supplied to it on a bus. Once the signal constituting the beginning of the new packet is asserted, the search process starts. It does not matter whether the circuit is busy with some other activity, insertion, for example. In case the circuit is busy with insertion of the entry, all the values that insertion acquired is saved in the temporary registers, and search starts immediately. Once it is finished, the insertion resumes from the point at which it was interrupted. Remember that an immediate search is a key to the wire-speed routing, which allows IP quality-of-service (QoS) implementation.

Figure 3.3 shows the block-diagram of hardware implementation of the forwarding part of the router. The lookup/update controller interfaces memory, where the trie is stored; microprocessor via microprocessor interface; and the circuits that are located before and after it.

The speed of the memory is a main factor affecting the speed of the lookup. Cost-efficient design can use slower memory. For IPv4, using 5 nanoseconds (nsec) of static random access memory (SRAM) requires only 35 nsec for a search in the worst case. For example, on an OC-192 interface of a IPv4 router, the smallest packet with size of 64 bytes corresponds to 51 nsec. Therefore, after the search is done, there are 16 nsec for other tasks, such as insertion or deletion of an entry and collection of statistical data. Deletion of an entry requires at most 46 memory cycles, or 230 nsec. Therefore, one insertion or deletion can be performed for every 15 lookups. This means that it is possible to perform more than 1.3 million updates per second, which is more than enough to satisfy the most demanding routing algorithms. These numbers are guaranteed for the worst case with respect to the length of the incoming packets and the location of the entries in the trie.

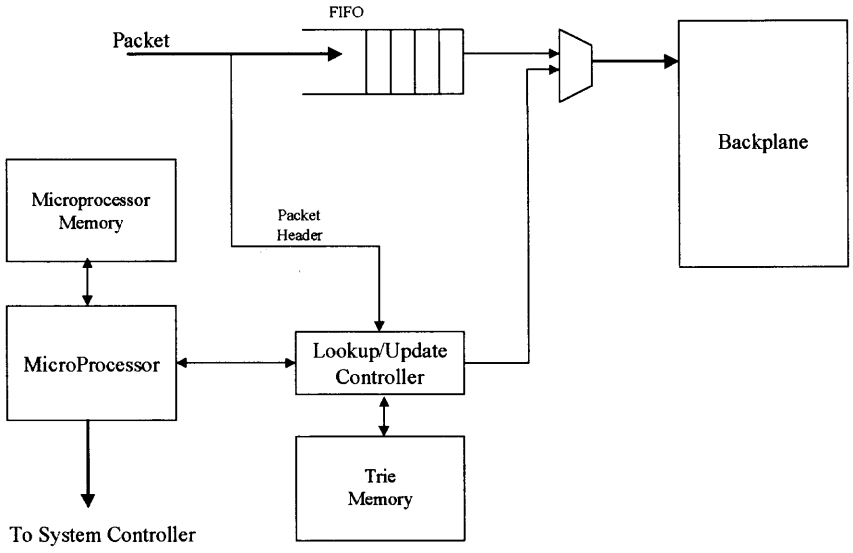

Figure 3.3 Block diagram of hardware implementation

Any general purpose microprocessor can be used with the controller. Depending on its characteristics, such as clock rate, width of the data and address bus, its interface has to be designed accordingly.

First In, First Out (FIFO) systems in the controller are used to store instructions from the microprocessor and to send acknowledgements back. Because the updates are not so time critical, they can be queued, unlike searches.

As mentioned before in the previous subsections, a good choice for values of K for IPv4 and IPv6 may be 4 and 8, respectively. In this section, the worst- and the best-case timing and memory requirements are presented. The worst-case requirements for timing are defined as the time necessary for the operation with a 29-bit prefix for IPv4 and a 121-bit prefix for IPv6. These require the longest time. The worst case for memory requirements occurs when all the entries are located in the individual nodes of the trie. Each node has only one valid child organizing the nodes as a number of linked lists.

Number of memory operations required for search is bounded by the depth of the trie, which are 7 for K=4, and 4 for K=8 considering IPv4. For K=4, worst case requires 27 memory operations for insertion and 46 memory operations for deletion.

3.5 Simulation of LLCAT

We have implemented LLCAT algorithm in C and tested with data collected from several Internet sources [6,12,18]. The implementation is not a software solution, rather a simulation of the hardware design of LLCAT. Therefore, time for operations is not relevant, however, number of memory operations is relevant. The code may be referred to in the Appendices section.

We have five routing tables. Aads, Mae-East, Mae-West are the main US exchange points provided by [12]. Vbns table is from the vBNS experimental network [18]. Funet table is obtained from [6], which comes from a European ISP and dates back to 1997. Memory consumption of the algorithm for the trie structure is given in Table 3.1 for different tables. Figure 3.4 gives the memory consumption for each of the five tables according to number of prefixes in the table. The drop at Funet table memory size at 41709 prefixes indicates that prefix distribution is more effective on memory consumption than the number of prefixes in the table.

Table 3.1 Routing tables used in simulations

Router	AADS		MAE-WEST		MAE-EAST		FUNET		VBNS	
Number of Prefixes	15566		28611		47196		41709		1816	
Average Prefix Length	22.05		21.88		22.04		22.02		21.01	
K	4	8	4	8	4	8	4	8	4	8
Number of Words	139,312	647,424	212,608	860,928	282,384	987,136	202,736	674,304	23,968	117,504
Memory Size (KB)	731	3,556	1,116	4,729	1,517	5,423	1,064	3,704	117	602

Table 3.2 Memory operation counts during table build-up (rows 6-11) and lookup with real packet data (last row)

Router	AADS					MAE-WEST					MAE-EAST				
K	1	2	4	6	8	1	2	4	6	8	1	2	4	6	8
Number of Nodes	48,038	21,994	8,691	7,400	2,528	76,656	34,813	13,272	11,777	3,362	108,278	48,219	17,633	16,011	3,855
Number of Words	96,332	88,232	139,312	473,856	647,424	153,568	139,508	212,608	753,984	860,928	216,812	193,132	282,384	1,024,960	987,136
Memory Size (KB)	494	452	731	2,545	3,556	806	732	1,116	4,142	4,729	1,138	1,014	1,517	5,630	5,423
Avg.Read	15.93	9.06	5.87	7.21	7.50	15.82	9.04	5.88	7.63	8.21	15.98	9.12	5.92	8.05	8.09
Avg.Write	6.05	4.31	3.70	8.94	9.18	5.43	3.96	3.56	9.12	9.43	4.86	3.61	3.34	8.85	8.49
Max Read	22	12	13	36	131	24	13	14	36	132	24	13	14	36	132
Max Write	26	17	15	38	132	27	17	17	38	132	27	16	16	37	132
Avg.Op	21.98	13.38	9.56	16.15	16.68	21.26	13.01	9.44	16.74	17.64	20.84	12.73	9.26	16.89	16.58
Max Op.	44	27	22	69	260	48	30	25	69	261	45	27	23	69	261
Avg. Read per Lookup with NJIT Trace	10.78	6.20	3.85	3.17	2.60	11.56	6.55	4.02	3.27	2.69	11.72	6.62	4.04	3.30	2.70

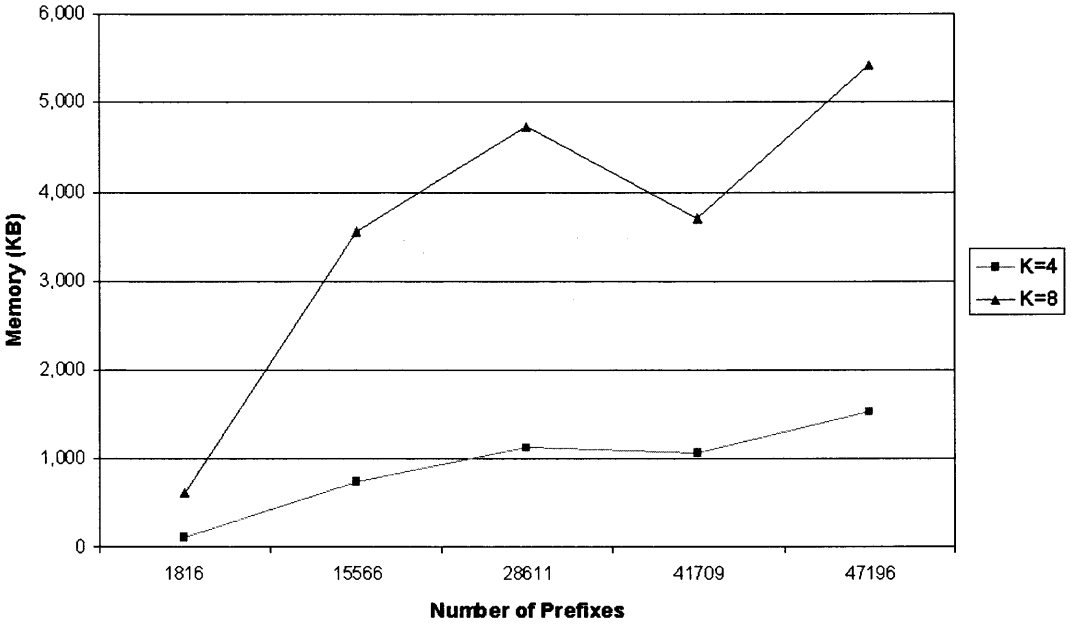

Figure 3.4 Memory consumption of LLCAT with respect to number of prefixes

We used packet traces obtained from New Jersey Institute of Technology network, and used it with Aads, Mae-East and Mae-West tables with different values for *K*. The results are summarized in Table 3.2. Table 3.2 shows average and maximum number of operation required for each prefix insertion during the build-up of the table, and in the last row, average number of read operations per look-up for the packets in trace. Trade-off between memory consumption and required number of accesses which determines the speed of algorithm is very apparent as can be seen in Figure 3.5, Figure 3.6 and Figure 3.7, which summarize results of Table 3.2. Notice that K=1 is indeed similar to a binary tree which provides one way branching, either left or right at each node depending of the value of the bit. It appears that K=4 provides an almost optimal point for memory and speed in the case of IPv4. Since most prefixes are 16 or 24 bits, it is important that K should be a divisor of 16 and 24

so that the words in trie nodes are better utilized. For updates, we have collected the next-day's routing table and extracted the differences and fed them in random order as update requests. Table 3.3 and Table 3.4 gives the memory operation counts for update operations.

We have observed that average case behavior of LLCAT, both in terms of memory operation counts and memory consumption turned out to be quite promising. It results that K=4 is suitable for IPv4, and K=8 may prove to be useful for IPv8.

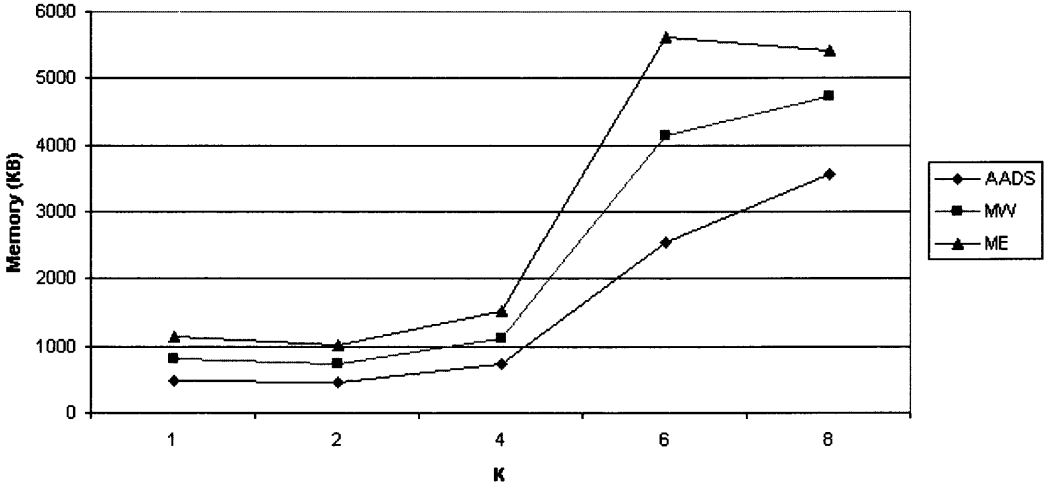

Figure 3.5 Memory consumption of LLCAT with respect to different values of K

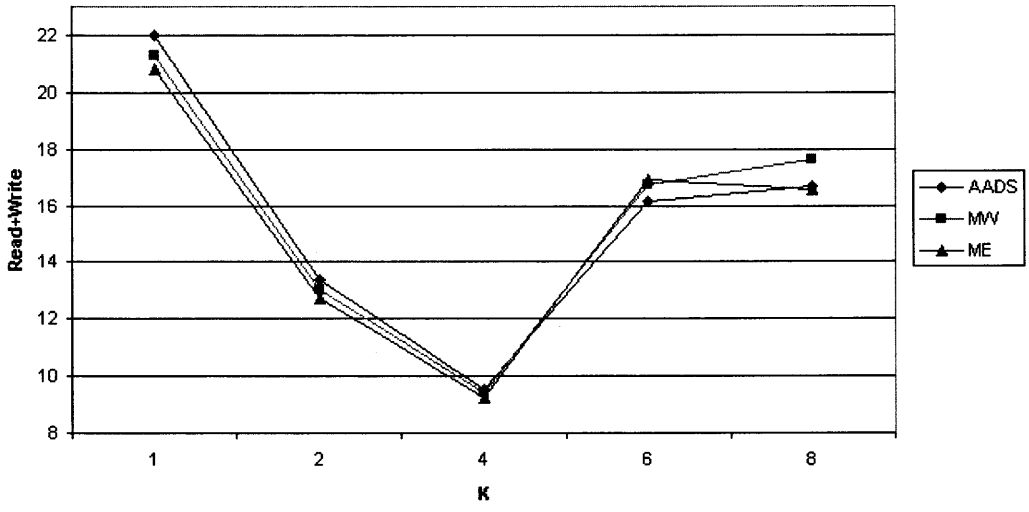

Figure 3.6 Memory operation counts during table build-up of LLCAT

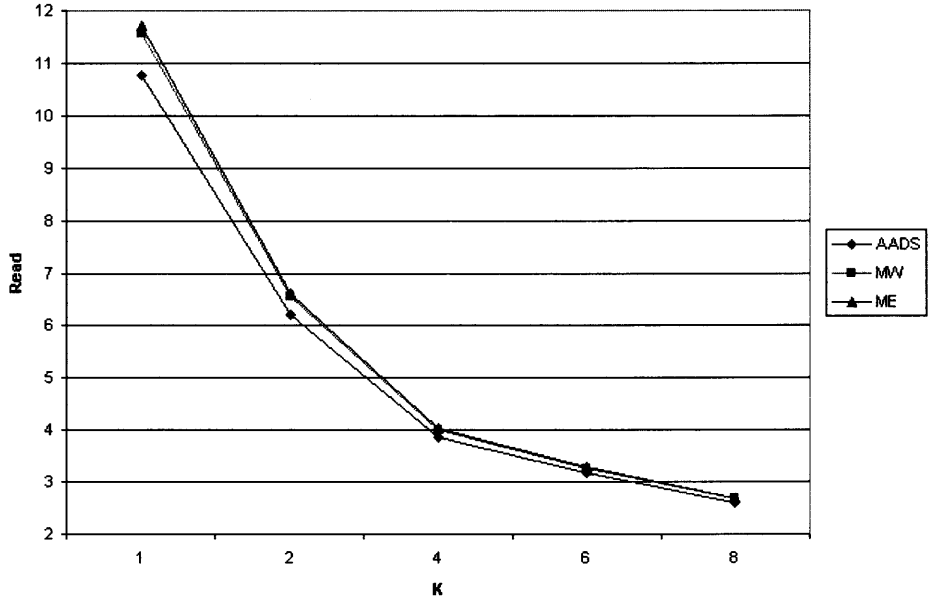

Figure 3.7 Memory read operation during lookup with real packet traces from NJIT network

We have also considered path compression as implemented in Patricia trie. For this, we have examined the trie structure for the existence of chains of empty nodes. Empty nodes are nodes which do not contain any routing decisions in the entries and

each empty node has only one forward pointer and therefore children count (*CC*) is 1. These intermediary nodes can be eliminated by introducing a pointer from the beginning of the chain to the last node of the chain, thus decreasing the search time and memory consumption. As depicted in Table 3.5, for values of $K \geq 4$ less than 6 per cent of the trie nodes were involved in chains, which makes the effort of skipping nodes unnecessary, simply because prefixes are already packed up in big nodes. For $K=1$ almost half of the nodes and for $K=2$ a quarter of the nodes were involved in chains, which may improve average look-up time at the expense of more complex search and update algorithms. Playing with the value of K appears to be a more sensible approach, which will definitely improve look-up time with a small increase in memory. If empty nodes are consecutive, meaning that the chain is long and if the prefix corresponding to the path represented by the chain is accessed frequently, then skipping nodes in the chain may be worth the effort in terms of decreasing average look-up time. We have also counted chains and nodes in the trie and the results are in Table 3.6. Furthermore, in order to quantify the effect of skipping, we ran the lookup simulation with an extra operation of decrementing read count each time an empty node was traversed. The results of this simulation presented in Table 3.7 indicate that there is not much gain on lookup time with this test data. We suspect that the effect may be more significant on smaller tables. However, we find that the observed behavior does not justify the effort required for implementing the scheme.

As a result of our simulations with IPv4 data we believe that $K=4$ provides an almost optimal compromise between memory consumption and number of memory

accesses required for insertions, deletion and lookup without altering the elegance and simplicity of the algorithm.

Table 3.3 Memory operation counts for incremental insertion for LLCAT

Router	AADS		MAE-WEST		MAE-EAST	
K	4	8	4	8	4	8
Avg. Read	5.96	16.50	5.59	10.83	5.57	11.94
Avg. Write	2.51	14.69	2.05	9.03	2.03	10.03
Max Read	12	130	13	131	13	131
Max Write	12	128	11	131	11	131
Avg. Op	8.48	31.20	7.64	19.86	7.60	22.03
Max Op.	20	258	22	259	22	259

Table 3.4 Memory operation counts for incremental deletion for LLCAT

Router	AADS		MAE-WEST		MAE-EAST	
K	4	8	4	8	4	8
Avg. Read	6.80	8.72	6.85	8.64	6.71	9.17
Avg. Write	3.64	6.82	3.53	6.23	3.31	6.67
Max Read	15	68	15	71	14	71
Max Write	16	68	14	65	12	65
Avg. Op	10.45	15.54	10.38	14.87	10.03	15.84
Max Op.	31	136	29	136	26	136

Table 3.5 Trie structure considering empty nodes that form chains

Router	AADS					MAE-WEST					MAE-EAST				
K	1	2	4	6	8	1	2	4	6	8	1	2	4	6	8
Total Number of Nodes in Trie	48,038	21,994	8,691	7,400	2,528	76,656	34,813	13,272	11,777	3,362	108,278	48,219	17,633	16,011	3,855
Number of Empty Nodes (where there is only one forward pointer)	23,670	5,838	551	93	8	33,026	7,245	512	62	12	40,349	8,075	526	60	9
Percentage Ratio	49.27%	26.54%	6.34%	1.26%	0.32%	43.08%	20.81%	3.86%	0.53%	0.36%	37.26%	16.75%	2.98%	0.37%	0.23%

Table 3.6 Trie structure considering chains and number of nodes involved in chains

Router	Chains with at least 2 Nodes			Chains with at least 3 Nodes	
	K=1	K=2	K=4	K=1	K=2
AADS	5,869	1,372	28	3,464	394
MW	8,219	1,541	23	4,514	351
ME	10,000	1,561	20	5,125	381

Table 3.7 Lookup operation counts when skipping is considered

Router	AADS					MAE-WEST					MAE-EAST				
K	1	2	4	6	8	1	2	4	6	8	1	2	4	6	8
Original Lookup	10.783	6.198	3.852	3.169	2.605	11.558	6.549	4.016	3.273	2.693	11.718	6.616	4.037	3.298	2.696
Lookup with Skipping	9.208	5.770	3.838	3.161	2.605	10.492	6.380	4.008	3.273	2.693	11.033	6.489	4.033	3.298	2.696

CHAPTER 4

COMPARISON OF ROUTING LOOKUP ALGORITHMS

There are two approaches that we can use to compare the performance of different algorithms. We can project available simulation results implemented in different platforms to a common platform and try to be as fair as possible by scaling memory access times and/or processor speeds. Table 4.1 is a summary of such an approach as depicted in [16] with LLCAT included. Or, we can try to evaluate on a theoretical basis using Big-Oh notation for complexity. Table 4.2 is a summary of such an approach by [17] with LLCAT included, where W is the length of the address, 32 for IPv4, N is number of prefixes, s is an algorithm dependent constant, K is the block size for LLCAT, and *n/a* meaning not available. For LLCAT, we have our own experimental results. Memory consumption is logarithmically proportional to number of entries in routing table for a specific value of K, around 1,5MB for 47,000 entries with K=4. Required read operations for lookup are not dependent on the number of entries, but the distribution of prefixes, and it is bounded by the depth of the trie, which is 7 for K=4, and 4 for K=8 considering IPv4. The experimental average case gives us lower bounds. Expected number of memory operations for update is around 8 for insertion and 10 for deletion for K=4, which is impressive.

Table 4.1 Lookup times for various schemes on a 300 Mhz Pentium II with 15 nsec 512 KB L2 cache

Algorithm	24 bit prefix lookup (nsec)	Worst case lookup (nsec)	Memory for 40,000 prefixes (KB)
Patricia Trie	1500	2500	3262
6-way search	490	490	950
Binary Search on levels	250	650	1600
Lulea Scheme	349	409	160
LC Trie	1000	n/a	700
Expanded Trie (leaf-pushed variable stride with 4 levels and packed array nodes)	206	n/a	450
LLCAT	5x15=75	7x15=105	1060

Table 4.2 Worst-case complexity of various algorithms

Algorithm	Insertion	Deletion	Lookup
Patricia Trie	n/a	n/a	$O(W^2)$
Dynamic Prefix Trie	O(N)	O(N)	$O(\log_k(N)+1)$
6-way search	O(N)	O(N)	$O(\log_k(N)+W)$
Binary Search	n/a	n/a	O(W/s)
Complete Prefix Trie (Lulea Scheme)	n/a	n/a	O(log W)
Binary Hash Table Search (Binary Search on Levels)	n/a	n/a	O(W/s)
Large Memory Architecture	n/a	n/a	O(W/s)
LC-Trie	n/a	n/a	O(W/s)
Expanded Tries	n/a	n/a	O(W/s)
LLCAT	$O(W/K+2^{K-1})$	$O(W/K+2^{K-1})$	O(W/K)

CHAPTER 5

CONCLUSION

For evaluating routing lookup algorithms, there are several criteria. Many schemes have been proposed especially in the last two years foreseeing the need for faster routing in Gigabit/Terabit networks, in which router performance becomes a major bottleneck. The relevant weights to each criteria are assigned uniquely for different environments and applications. Each scheme has its own strengths and weaknesses.

We have proposed a trie based algorithm which we call LLCAT, with particular strength in update operations and ease of implementation in hardware. Simulation results suggest that LLCAT will meet the performance demands of high speed line-rate lookup and heavy loads of update requests. We believe that such demands may be only met by sophisticated hardware solutions. Demands will increase with the introduction of wireless networks with mobile subnets.

For less demands for speed and light loads of updates, simple software solutions will suffice. We believe that routing instabilities do not impose much load on the router at the enterprise level, compared to backbone level. Routing lookup with frequent updates will continue to be a bottleneck for high performance routers. Introduction of protocol based solutions does not eliminate the need for routing lookup algorithms. Indeed, lookup algorithms are incorporated into such solutions.

APPENDIX A

DETAILED DESCRIPTION OF THE ALGORITHM

In this appendix, the algorithm will be explained in detail by moving from simple examples to more complex ones to demonstrate the intricacies involved in procedures.

```
1000 0000 0000 1010 0000 0010 0001 *  →   X

  128   .   10    .    2    .  16 / 28   →   X
```

The routing tables consist of prefixes each of which represents a network address, and its corresponding routing decision as shown above. Number of bits in the prefix are common to all networks and hosts specified by this range. It is convenient to represent the prefix in decimal notation like IP numbers and specify the number of bits in the prefix as shown in the example above for a prefix of length 28.

For a short table, it may be sufficient to make a linear search of all entries in the table. However, this is not feasible with tables of more than a thousand entries. Special data structures for storing and searching have been developed. The most popular data structures are variants of tries. Trie is a tree where the traversal path is associated with the information stored. The idea of LLCAT evolves from the simplest trie, the binary trie. The binary trie shown in Figure A.1 has 3 leaves each of which corresponds to 2 bits of information because each level traversed down the trie corresponds to one bit, 0 if traversal follows left branch, 1 if traversal follows right branch.

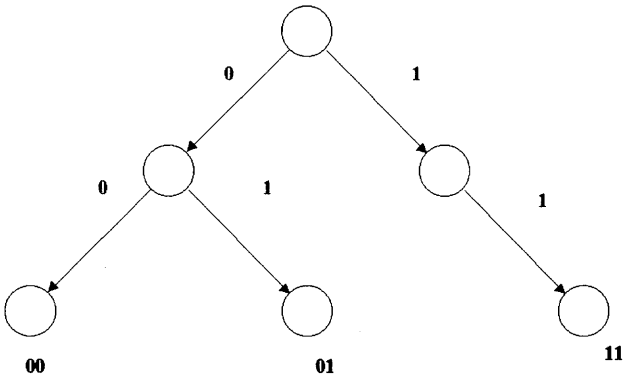

Figure A.1 A simple binary trie

We can use this trie to store prefix information and look up routing decision. Search procedure will examine one bit of the IP address and will follow left branch if it is 0, right branch if it is 1. Then the maximum depth of the trie will be 32 and the search will read 32 nodes at most during traversal.

If we can allow more than two outgoing pointers from each node then we can represent more than one bit at each level of the trie. Figure A.2 shows a 4-ary trie, where each level represents 2 bits of information. For this trie, the search will examine 2 bits of the IP address at a time. The search will stop when it reaches the last level of the trie and return the routing decision at the last node it has traversed. Note that the depth of the trie is 32/2=16 for a 4-ary trie and the search will take time half of the binary trie.

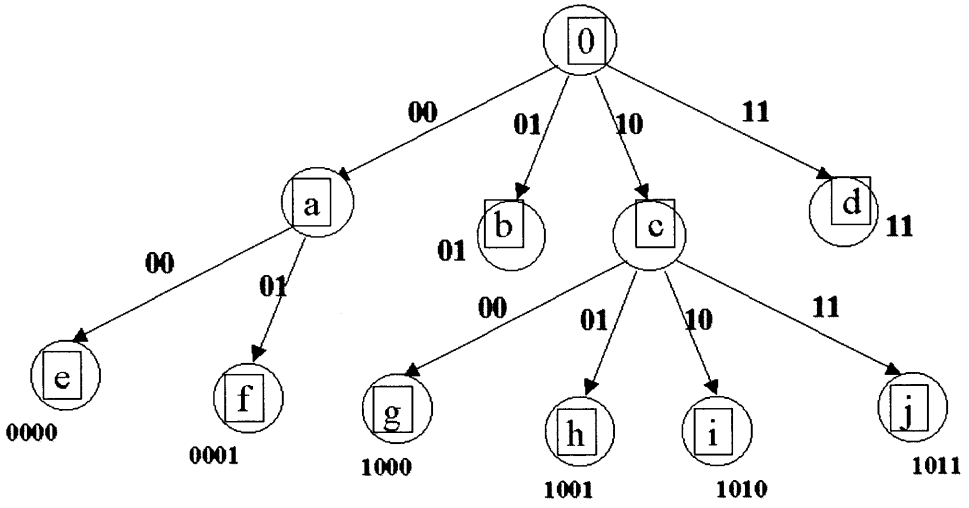

Figure A.2 A simple 4-ary trie

The trie has a nicer structure now. When we want to implement this structure in the memory, we can use any programming language and implement it on any general microprocessor by using records as nodes where each record will contain one routing decision field and four pointers to its children. The memory management will be handled by the processor. This is not the only way to implement this structure. If we want to take advantage of specially designed hardware and we have a memory array of words which we can freely manipulate, then we can implement this structure in a more efficient and straightforward manner. First consider that each node has four children. If we merge all children into a single node, we will obtain a structure shown in Figure A.3. Both tries in Figure A.2 and Figure A.3 are referred to as multi-bit tries.

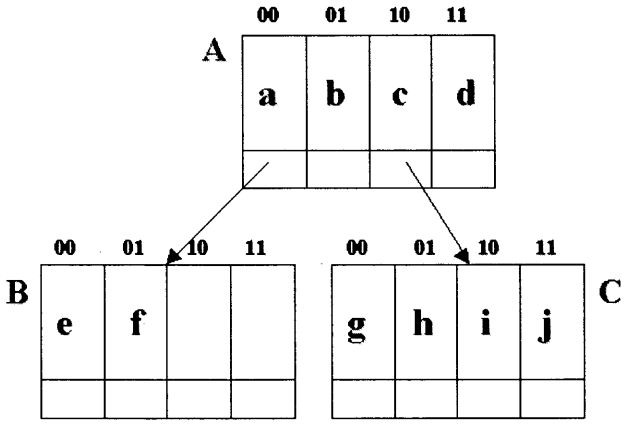

Figure A.3 A simple 4-ary trie with merged nodes

In this trie, note that the four children of a node have become entries/words in a single node, and these entries are placed continuously in the node. If we think of the node as a segment in memory, then the entries are given by their offsets in the segment. Thus, we will efficiently store the trie in the memory array as in Figure A.4. Note that the outgoing, forward pointers point to the node/segment. We can use this structure for storing and looking up routing decisions. We have to know the address of the root node of the trie. At each level, we will extract 2 bits from the IP address and use it as offset in the current node to find the related entry. In this way, we can devise a scheme to manage the memory and keep track of number of memory operations.

A 00	a	B
A 01	b	-
A 10	c	C
A 11	d	-
B 00	e	-
B 01	f	-
B 10	-	-
B 11	-	-
C 00	g	-
C 01	h	-
C 10	i	-
C 11	j	-

Figure A.4 The representation of the trie in the memory

Note that each node in the trie corresponds to 2 bits in traversal path. We call this value *K*. For brevity, the following examples will use tries where *K*=2 and there are 4 entries in the node.

This structure is useful for look up operations and it can be easily generated from a given routing table. However, a structure that supports incremental insertion and deletion is desirable and the structure as it is shown in Figure A.4 is not appropriate for these purposes. The reasons for this problem and its solution is presented below.

Insert 001*→ n

When a new prefix is inserted, the procedure first traverses the trie to the last level. In this example, the first two bits are 00, so we follow the pointer at first entry of node A to get to Node B. The next bits of the prefix is 1*. When we expand the prefix to 2

bits which is the coverage of the node, we have the entries 10 and 11 where the routing decision for the prefix is to be inserted. Then routing decision "n" is written to entries 10 and 11 of Node B. The resulting trie is in Figure A.5.

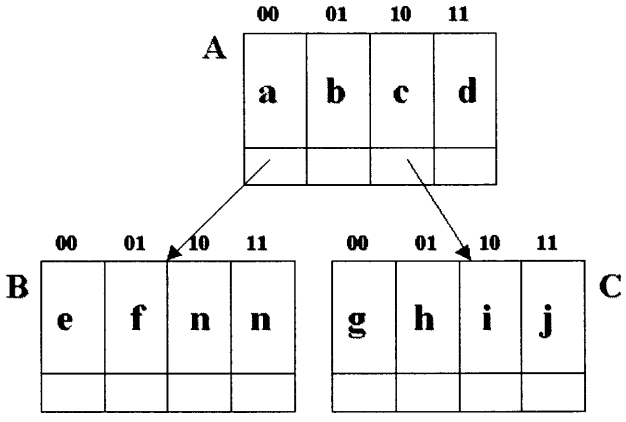

Figure A.5 The trie after insertion of 001*→n

Insert 0011*→ m

To insert routing decision "m" we traverse to Node B. This time, last two bits are 11 and there is no need for expansion because we have a full prefix. We insert the routing decision in the entry 11 directly. There is a previously inserted routing decision in this entry. However, since it represents one bit information and we have a full prefix which represents a longer prefix, we are allowed to override the previous information. So far, we have not encountered any problems with insertion. The resulting trie is in Figure A.6.

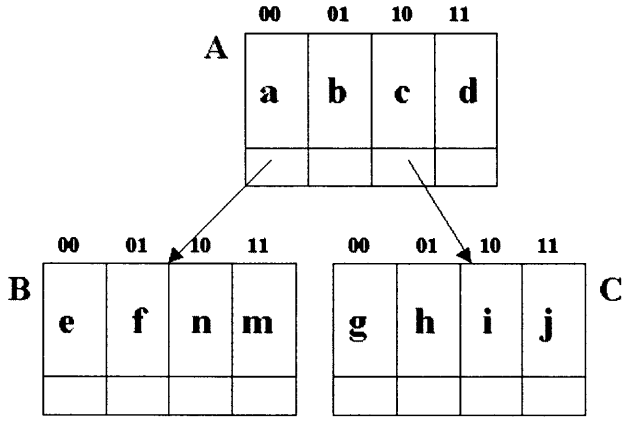

Figure A.6 The trie after insertion of 0011*→m

Delete 0011*→ m

When we want to delete the prefix that we have just inserted, we do the reverse operation that we did for insertion. We go the entry 11 in Node B and we delete the routing information. However, this time we have insert the routing decision "n" which was overridden before. To find the related information we have to consider the original table or we have to store the information that was lost during the expansion in the trie structure somehow. The procedure has to be able to reconstruct trie of Figure A.5 from the trie of Figure A.6 upon deletion of the prefix.

Insert 101*→ z

When we insert the prefix 101* in Figure A.6, we traverse to Node C this time. We discover that the entry that was previously inserted is for a full prefix and we are not allowed to override its routing decision. However, if any of the entries 10 or 11 in node C is deleted, we must be able to write the decision "z" for these entries which are covered by the prefix 101*.

To store the original information of prefixes that are lost due to extension we introduce several fields to be stored in the entries as shown in Figure A.7.

S field is a bit flag which signals if there is a valid forward pointer at the FP field of the entry to follow to the next level of trie. If S=1, then the traversal stops at this node, otherwise, it moves onto the next level. R contains the routing decision for the longest prefix that covers the entry. When a routing decision is stored information is stored, to designate how long a prefix this information stands for, we set bits in valid sub-prefix pattern (VSP) field. For example, if a full prefix is inserted then the most significant bit of VSP is set. If the information to be inserted is for a shorter prefix than a previously inserted entry then the insertion procedure is not allowed to override information but the VSP bit is still set for the shorter prefix. Then, when the longer prefix is deleted we have a way of knowing the presence of a shorter prefix.

We also have to keep track of how many entries actually contain information in the node so that if there is no information we are able to delete the node from the trie. Information is a routing decision and/or valid forward pointer stored in the entry. We use children counter CC field for this purpose. We use the first entry of the node to contain the children counter for the node. In entries other than the first this entry has another purpose.

When we insert a shorter prefix that is not allowed to override a present routing information, we store the actual routing information for the unexpanded prefix in the CC/RS field of its "home entry". The address for the home entry of a sub-prefix that has L bits where L<*K* and K is the number of bits covered by the node is computed with the following steps:

- pad the sub-prefix with zeros to make it K bit long,
- set the L+1 bit from the left of the prefix to be 1.

For example, home entry of prefix 01* for K=4 is 0110. Thus, we can distribute sub-prefixes over the node and store their information in CC/RS fields. For a K-bit node we have exactly 2^K-1 sub-prefixes. All entries except 000... and 100... can accommodate the decision for a sub-prefix. Since the entry 000... is the first entry in the node and CC field is used as a child counter, we are wasting only on CC/RS field which remains unused in all the node.

When the deletion procedure deletes a longer prefix and finds out that there have been shorter prefixes inserted for the entry, it has to find the next longest prefix and insert its routing decision for the entry. The home entry of the shorter prefix is computed and the decision in its RS field is inserted in the R field of the deleted entry.

With the additional fields that we have added, Figure A.7 gives the resulting view of the memory for the trie of Figure A.6 after insertion of the prefix 101*→ z.

	S	VSP	R	CC/RS	FP
A 00	0	10	a	4	B
A 01	1	10	b	-	-
A 10	0	10	c	-	C
A 11	1	10	d	-	-
B 00	1	10	e	4	-
B 01	1	10	f	-	-
B 10	1	01	n	-	-
B 11	1	11	m	n	-
C 00	1	10	g	4	-
C 01	1	10	h	-	-
C 10	1	11	i	-	-
C 11	1	11	j	z	-

Figure A.7 The trie representation of the memory after insertion of prefixes, 001*→ n, 0011*→ m, 101*→ z to the trie of Figure A.7

Insert 00010*→ x

When a prefix is being inserted, the traversal follows to the last level of the trie by extracting bits of the prefix. If the traversal stops when a stop bit equal to 1 is encountered and there is no further nodes to follow, but bits in the prefix are not exhausted, this will mean that the depth of the trie must be increased to accommodate the new prefix. Then a new node is requested from the linked list of idle nodes. And it is appended to the last node of the trie encountered.

For the insertion of 00010* into the trie of Figure A.4, the traversal will go to nodes A and then B and will see that there is no further level below Node B. Then a new node D is acquired from ILL, forward pointer at Node B is set accordingly and

the remaining part of the prefix 0* is expanded into 00 and 01 and inserted in Node D. The resulting trie is in Figure A.8.

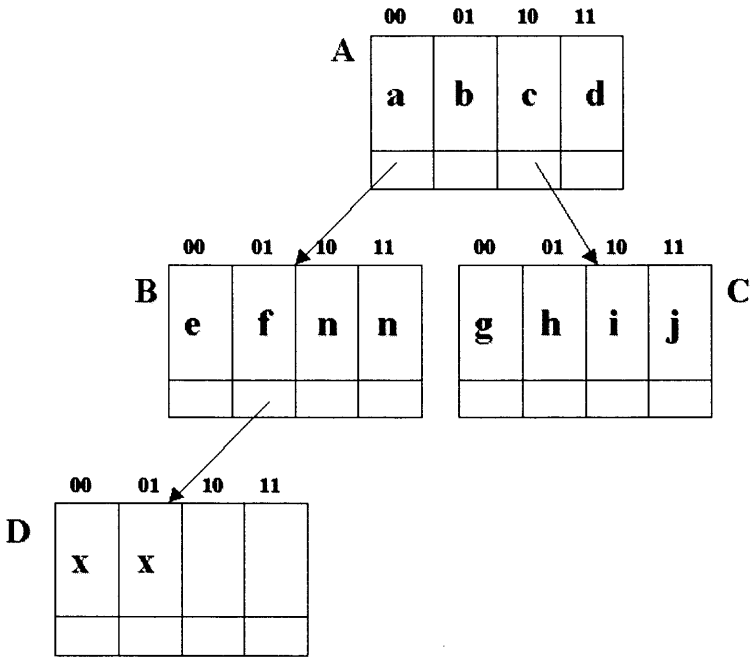

Figure A.8 The trie after insertion of 00010*→x

Idle Linked List (ILL)

Linked list of idle nodes (ILL) requires only two additional words in memory wide enough to hold segment address only. One word will be the head pointer (ILL_HP) and will contain the segment which is the head of the linked list. The other word will be the tail pointer (ILL_TP) and will contain the segment which is the last node in the linked list. The pointers in the linked list which will be pointing to the next node in the list will be stored in the forward pointer field of the first entry of each node. The logical view of the list is given in Figure A.9.

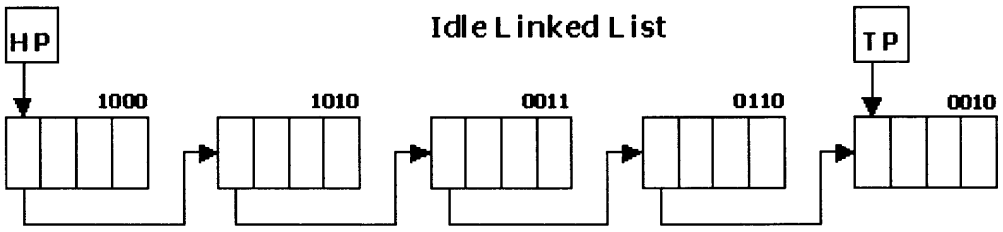

Figure A.9 Logical view of idle linked list

Choice of K

We refer to the design parameter *K* as the stride size. It is a very important parameter that determines the performance of the algorithm. The node size is 2^K and it determines the amount of memory consumed by the table. The depth of the trie is deterministically given by the value of *K*. During the search operation one memory read operation is performed for each stride. If we take into account that there are no prefixes less than 8 bits in today's routing tables and therefore make the first stride to be 8 bits, then we can calculate the required number of read operations for a 32-bit address lookup as given below.

K=1 $\Rightarrow$ Node size = 2, 32-8+1 = 25 Reads for Lookup

K=2 $\Rightarrow$ Node size = 4, (32-8)/2+1 = 13 Reads for Lookup

K=4 $\Rightarrow$ Node size = 16, (32-8)/4+1 = 7 Reads for Lookup

K=6 $\Rightarrow$ Node size = 64, (32-8)/6+1 = 5 Reads for Lookup

K=8 $\Rightarrow$ Node size = 256, (32-8)/8+1 = 4 Reads for Lookup

K can be any value in the range 1 through 32. In the large routing tables it is observed that the majority of the prefixes are concentrated at 16 and 24 bits. So it is

appropriate to choose K to be a divisor of 16 and 24. Then most prefixes will not have to be expanded and memory is better utilized.

Note that K=1 requires that there are two outgoing pointers at each node, therefore, the structure is very similar to binary trie. Also note that K=32 corresponds to a direct memory lookup on the IP address.

Skipping nodes in search

To improve search time a shortcut to the lower levels of the trie may be introduced as shown in Figure A.10. In order for the search operation to follow this shortcut, we have to be sure that the short refers to the pattern we are looking for. Therefore, the pattern for which the skipped nodes stand and its length must be stored. Then the search will not read the entries in skipped nodes, it will directly read the entry in the final node. To store the extra information, and to modify search, deletion and insertion procedures increase both space and time complexity of the algorithm. Skipped nodes may not be inserted in the trie at all to save memory, but then, when a prefix pattern corresponding to the skipped nodes is to be inserted, all eliminated nodes have to be created again.

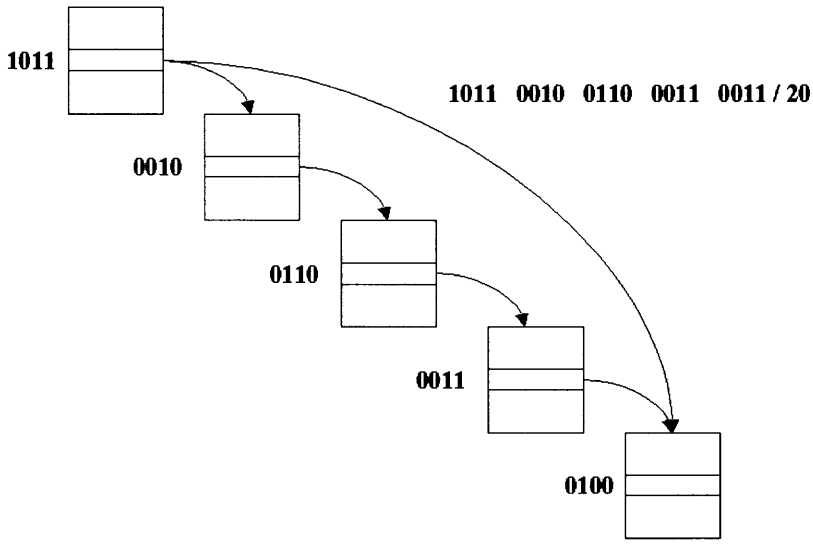

Figure A.10 Skipping nodes in search

APPENDIX B

FLOWCHARTS

In this appendix, flowcharts for the complete algorithm are provided. Please note that the flowcharts describe the algorithm where the IP address is partitioned into K-bit blocks. It does not take into account that the root node of the trie contains 256 entries and corresponds to 8 bits regardless of the value of K as it was implemented in the simulation code for all values of K.

We have utilized some conventions in the flowcharts. Double-lined rectangles designate where a memory operation has occurred. We have used temporary variables such as loop control counters. The variable "Word" corresponds to an internal register having the width of a word in the memory array and bits are used as fields. We will refer to fields with the notation "Word.*fieldname*". Root of the trie corresponds to the address of the first node of the trie at level 0.

Initialization of a word refers to setting all bits to 0 except the bit for S field which is set to 1.

As clearly noticed in the flowcharts, the algorithm contains the following types of operations:

- bit-wise operations (shift, and, or),
- addition and subtraction,
- memory read and write,
- conditional branching based on equality check.

Search

Search procedure is given in Figure B.1 Search procedure on the trie. The procedure returns the routing decision for the longest prefix that matches the address IP. *K* bits of the IP address are examined and used as offset in the current node at each iteration of the loop. The procedure ends when a stop bit set to 1 is encountered which may happen at any level of the trie. If a default routing decision exists, then the variable "match" should be assigned the default value before entering the loop.

Insertion

Insertion procedure is given in Figure B.2 through Figure B.6. It inserts a prefix P having (M×*K*+L) bits and its corresponding routing decision RX. The procedure starts with traversal of the trie to the last node as depicted by the last bits of the prefix in Figure B.2. If a stop bit is seen to be set before bits in prefix P are exhausted, then a new node is appended to the trie. For a new node, the entries on traversal path need not be read because they are known to be empty. This is signaled by the Boolean Flag NewNode. If a new node is created and forward pointer is attached to a previously invalid entry then children counter of the node has to be incremented. If the offset is 0 then children counter (CC) is already available in RS field and it can be written directly after incrementing. Otherwise, the first entry of the node has to be read, RS field is incremented and written back. This is Part 1 of the insertion procedure. Details of getting a new node from the linked list of idle nodes is given in Figure B.6. Now, we are in the last level of the trie where the actual prefix information has to be stored.

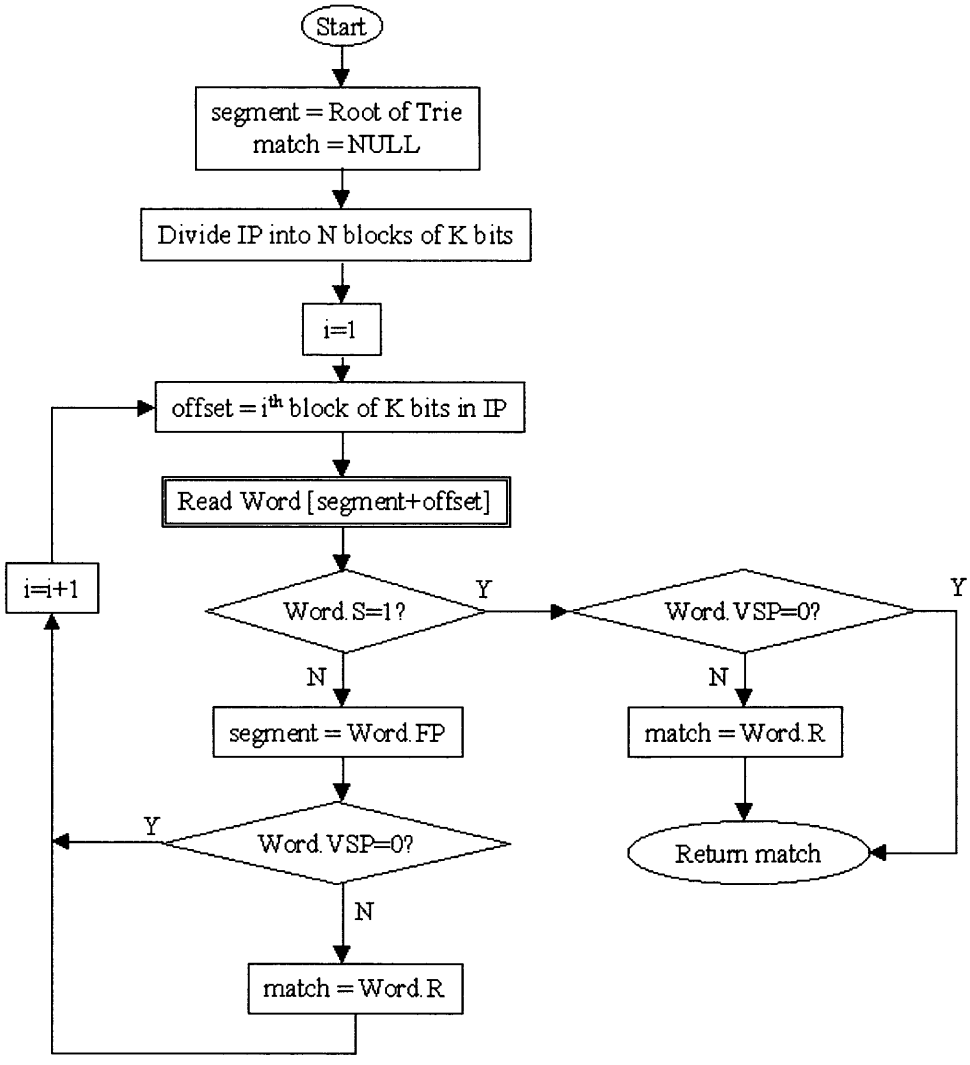

Figure B.1 Search procedure on the trie

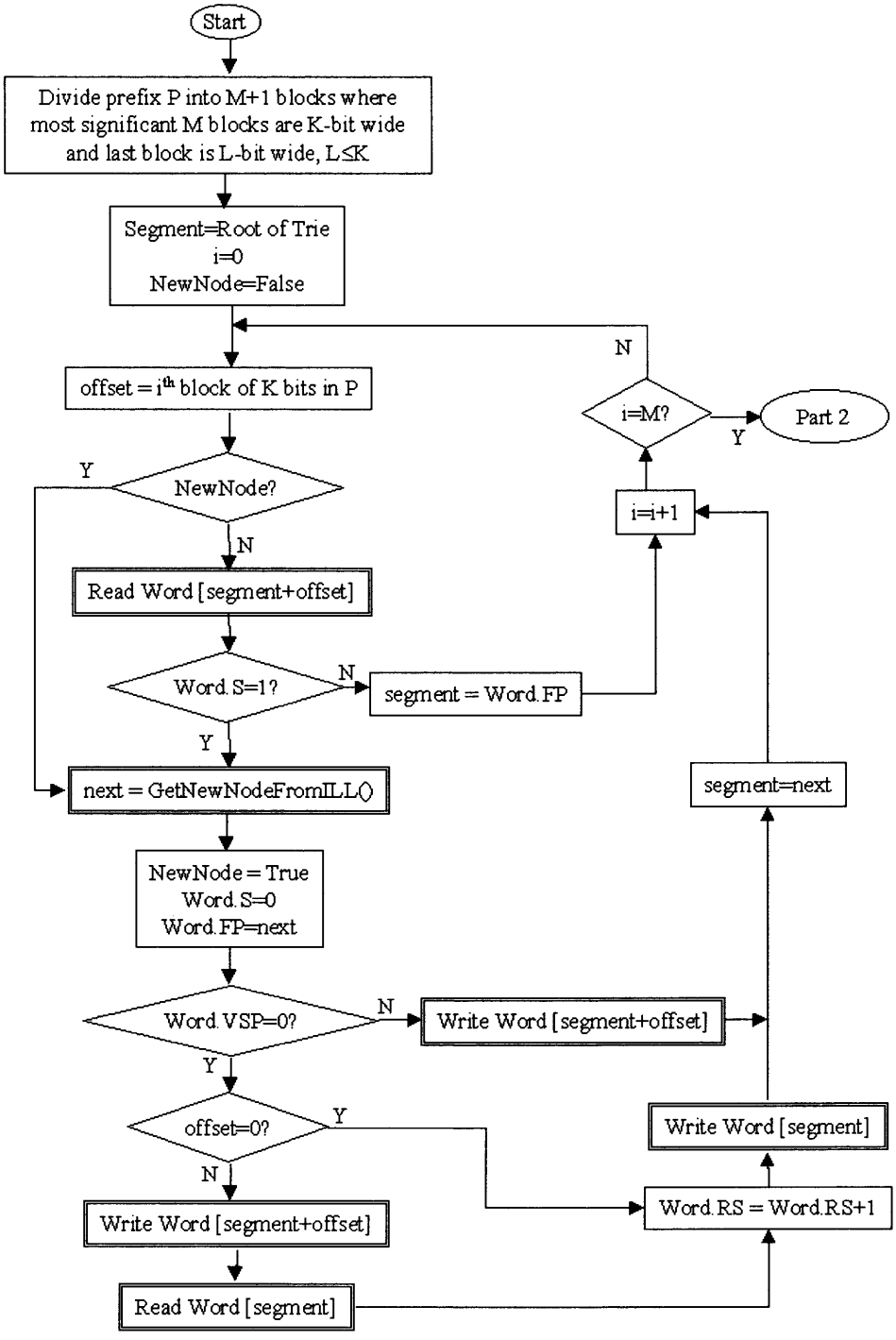

Figure B.2 Part 1 of Insertion procedure on the trie, traversal to the last level

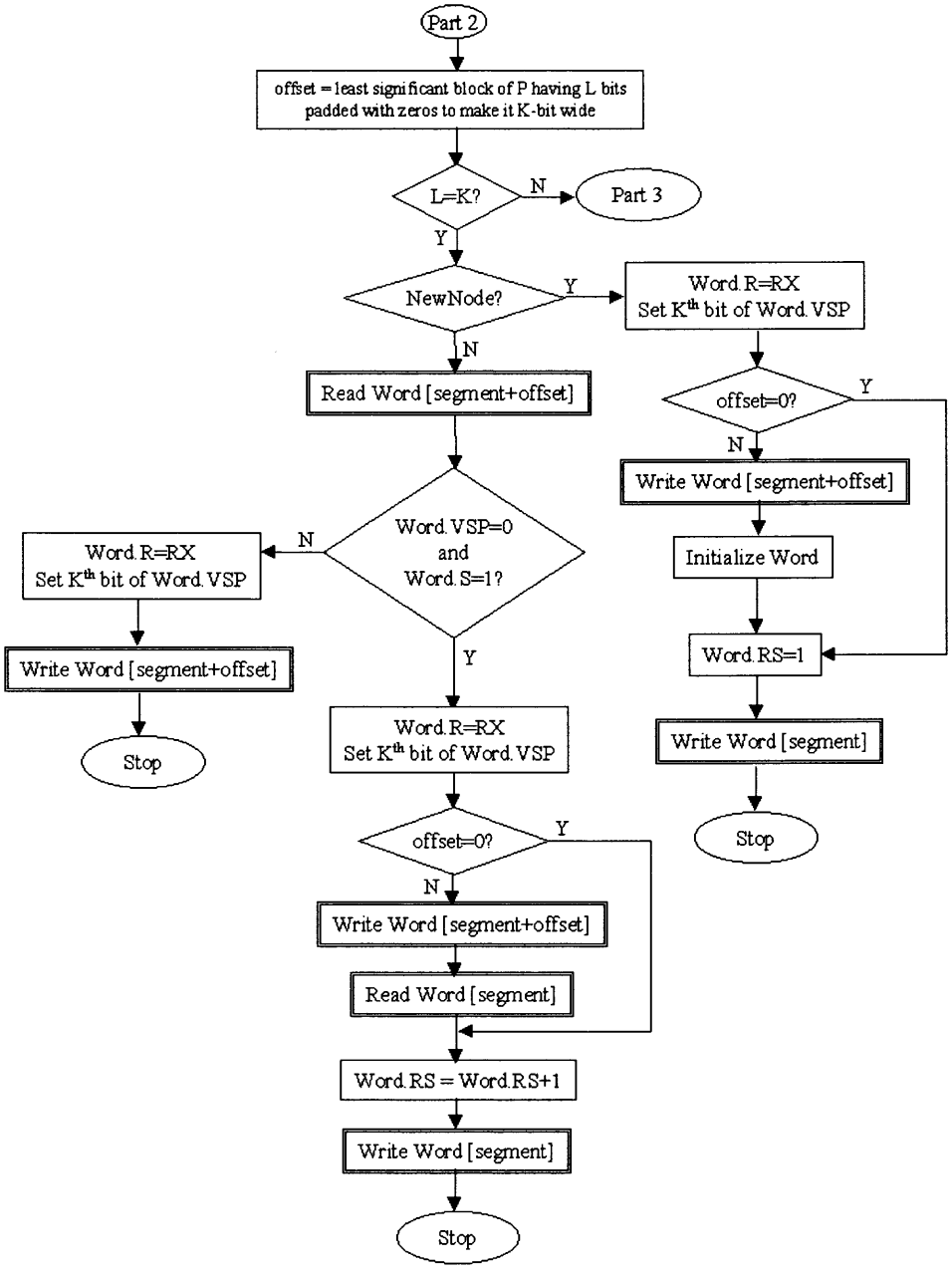

Figure B.3 Part 2 of Insertion procedure on the trie, inserting a full prefix

We extract the address of the offset from the last L bits of the prefix. If L=*K* then we are inserting a full prefix and there is no need for expansion. The insertion will affect only one entry in the node. If it is a new node, it is going to be the only entry in the node. Therefore, we can set the children counter to be 1. If it is not a new

node, and the entry is not valid, then we have to increment the children counter. As in Part 1, if offset=0 then there is no need to read the word for children counter since it has already been read. If it is not a new node and the entry is valid, then children counter does not change. We write back the routing decision in the R field, and set K^{th} bit to 1 of the entry affected by the prefix. Since there can be no longer prefix, we can override the routing decision if it has been previously inserted.

If the prefix is not a full prefix, i.e. $L \neq K$, then the prefix will be expanded to cover all entries affected. In this case, more than one entry will be affected. These entries will be consecutive entries in the node so they can be referred to by decrementing the memory address starting from the last affected entry in the node. While evaluating the entries we keep track of new entries so that children counter can be added the number of new entries. We start evaluating the entries from the last node down to the first affected entry. The reason is that the first affected entry may also be the first entry in the node. When ready to write back the first entry we will have acquired information about how much children counter has to be increased and we can write it immediately. If we do not start from the last entry, we will have to write the first entry twice if it is also the first entry in the node. Since it is not a full prefix, we also have to insert the routing decision into the RS field of home entry of the sub-prefix whose address is computed as H for offset in the node.

If it is a new node, entries will be empty and there is no need to read. Corresponding fields will be adjusted and written directly. The routing decision will be inserted in all of the R fields since there is no previous routing information in the node. This is Part 3 of the insertion given in Figure B.4.

If it is a not new node, then we will have to read the entries and check the routing information that has been previously inserted. The routing decision is not going to be inserted in entries where a routing decision for a longer prefix has been inserted previously. Since the prefix covers L bits, it is sufficient to check if any bits higher than L has been set. If not, only then the routing decision is allowed to overwrite the previous decision. We keep track of new entries in the variable "CC_increment". After all entries have been evaluated, if CC_increment≠0 then the children counter is increased by that amount. This is Part 4 of the insertion given in Figure A.5.

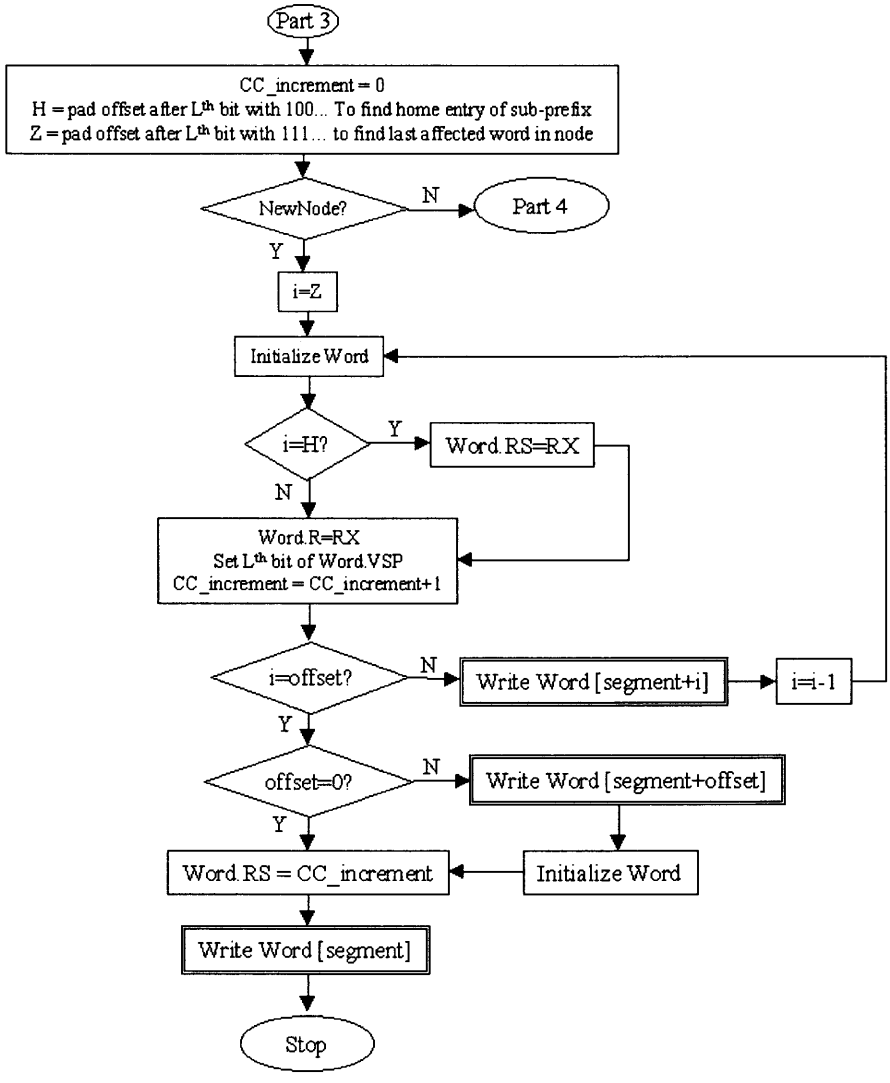

Figure B.4 Part 3 of Insertion procedure on the trie, inserting a sub-prefix into a new node

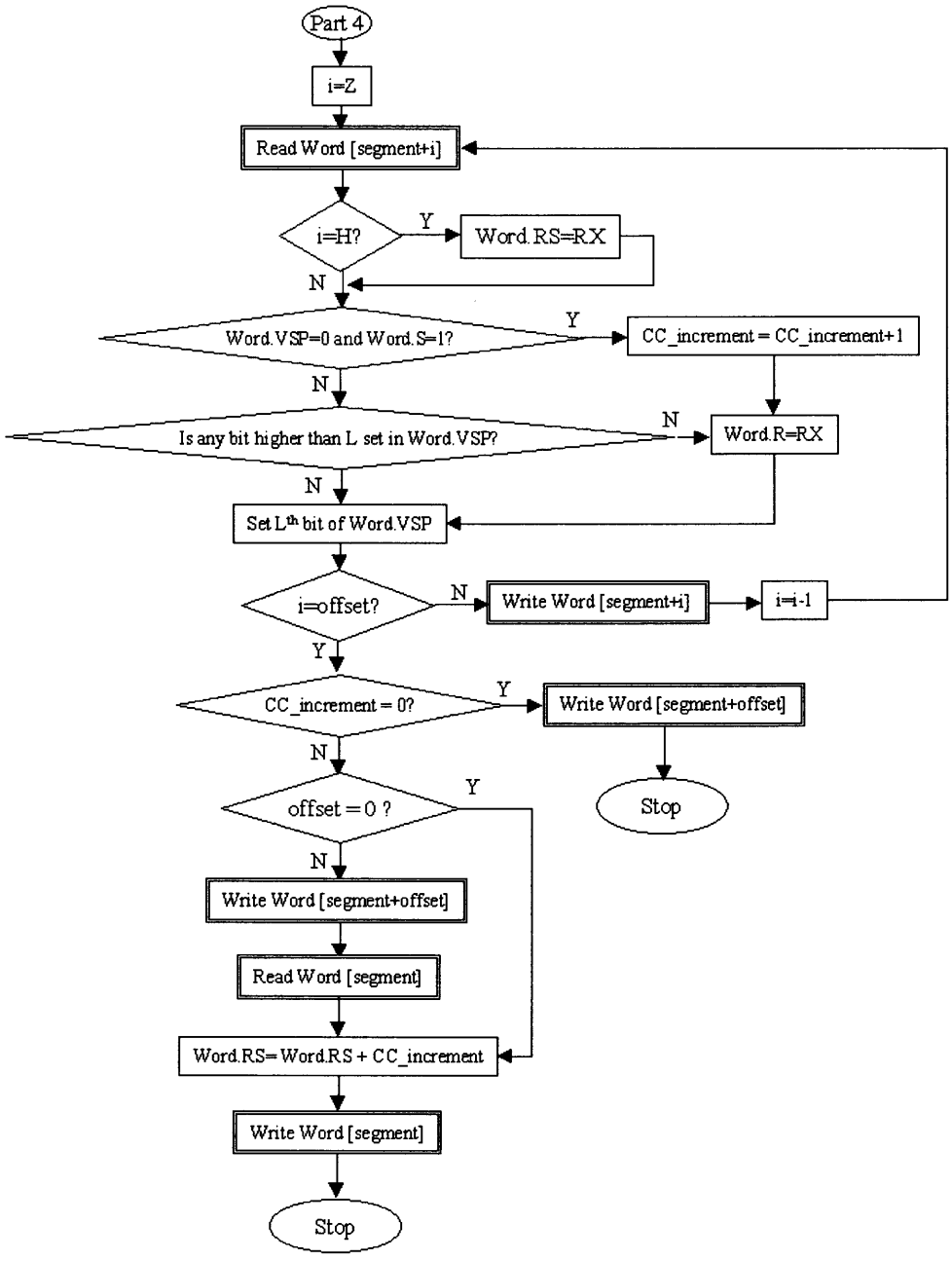

Figure B. 5 Part 4 of Insertion procedure on the trie, inserting a sub-prefix into an old node

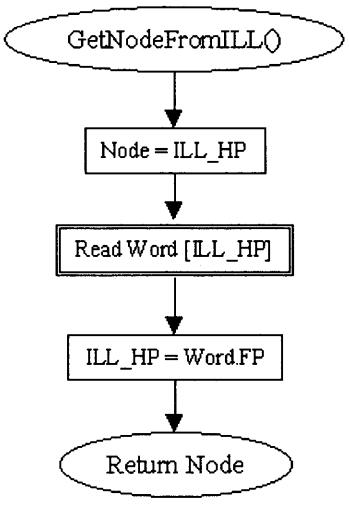

Figure B.6 Getting a new node from the linked list of idle nodes

In Figure B.6, details of getting a new node from the linked list of idle nodes (ILL) is given. The value of the head pointer is returned as the new node. It involves only one read operation to update head pointer of the list (ILL_HP). Note that Word variable used in GetNodeFromILL has to be different from Word variable used by insertion procedure, because it is overridden in GetNodeFromILL but referred to again in insertion before any other assignment is made.

When a new node is acquired from ILL, it is known that all entries in the node are in initial state where S=1 and all other bits are zero. This is guaranteed by the initial state of the memory and deletion procedure which we will examine in the next section.

Deletion

Deletion procedure is given in Figure B.7 through Figure B.12. It starts with traversal to the last node just like the insertion procedure. It deletes a prefix P having

(M×*K*+L) bits. At each level of the trie K bits are extracted from the prefix P and used as offset in the current node. If a stop bit is seen to be set before bits in prefix P are exhausted, then this means that the prefix to be deleted does not exist in the trie and the procedure is aborted. While traversing down the trie, all entries in the traversal path and their addresses are pushed to the deletion stack. We have to do this in order to keep parent information of the nodes which will be necessary if we delete the last node in traversal. If the last node is deleted, then its parent may be deleted as well, if the only information parent contains is the forward pointer to the deleted node. This procedure may follow through all the nodes in the traversal path up to the root of the trie. The procedure is maintained such that root node of the trie is never deleted. If a node is deleted, then all entries are set to the initial state and the node is appended to linked list of idle nodes (ILL). Details for adding a node to ILL is given in Figure B.12. Deletion of a full prefix and sub-prefix are different. If L=K then we have a full prefix to be deleted, otherwise it is a sub-prefix. This is Part 1 of the procedure given in Figure B.7. Deleting a full prefix is Part 2 given in Figure B.8, and deleting a sub-prefix is Part 3 given in Figure B.9. Now, we are in the last level of the trie where the prefix information is stored.

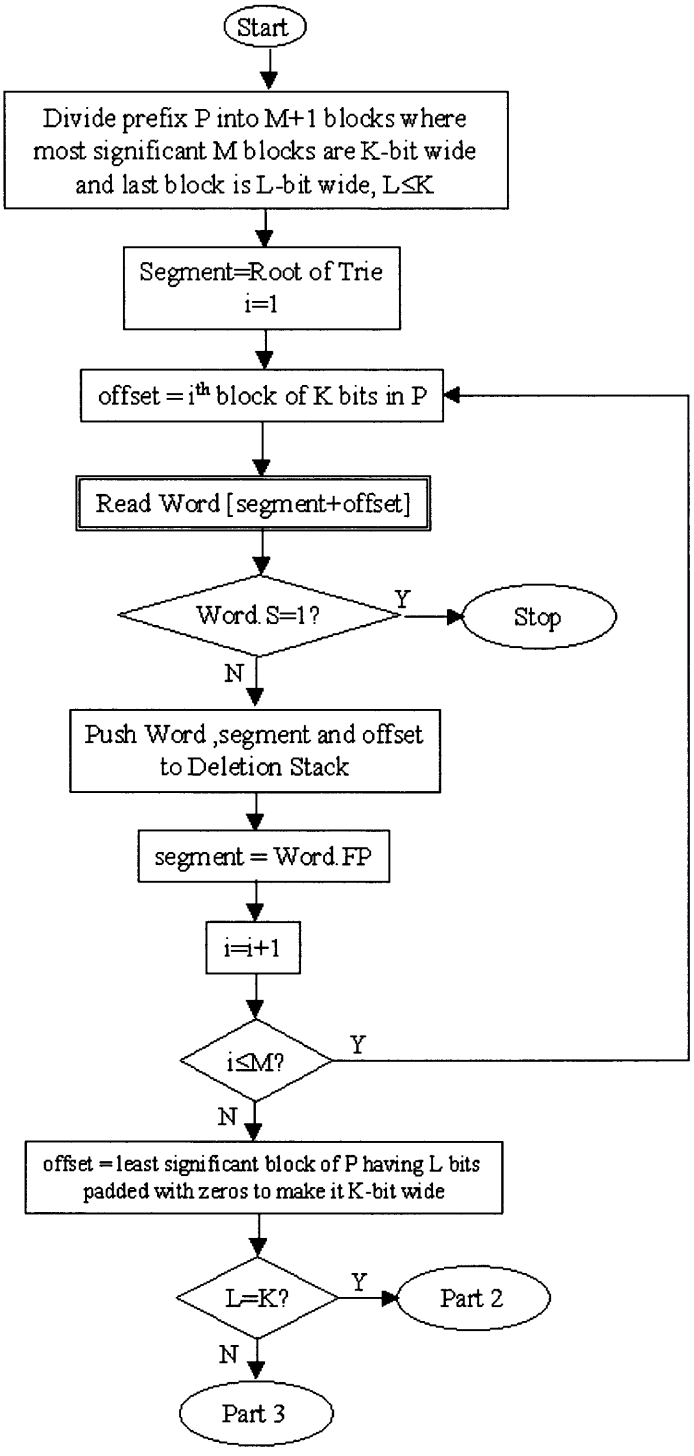

Figure B.7 Part 1 of Deletion procedure on the trie, traversal to the last level

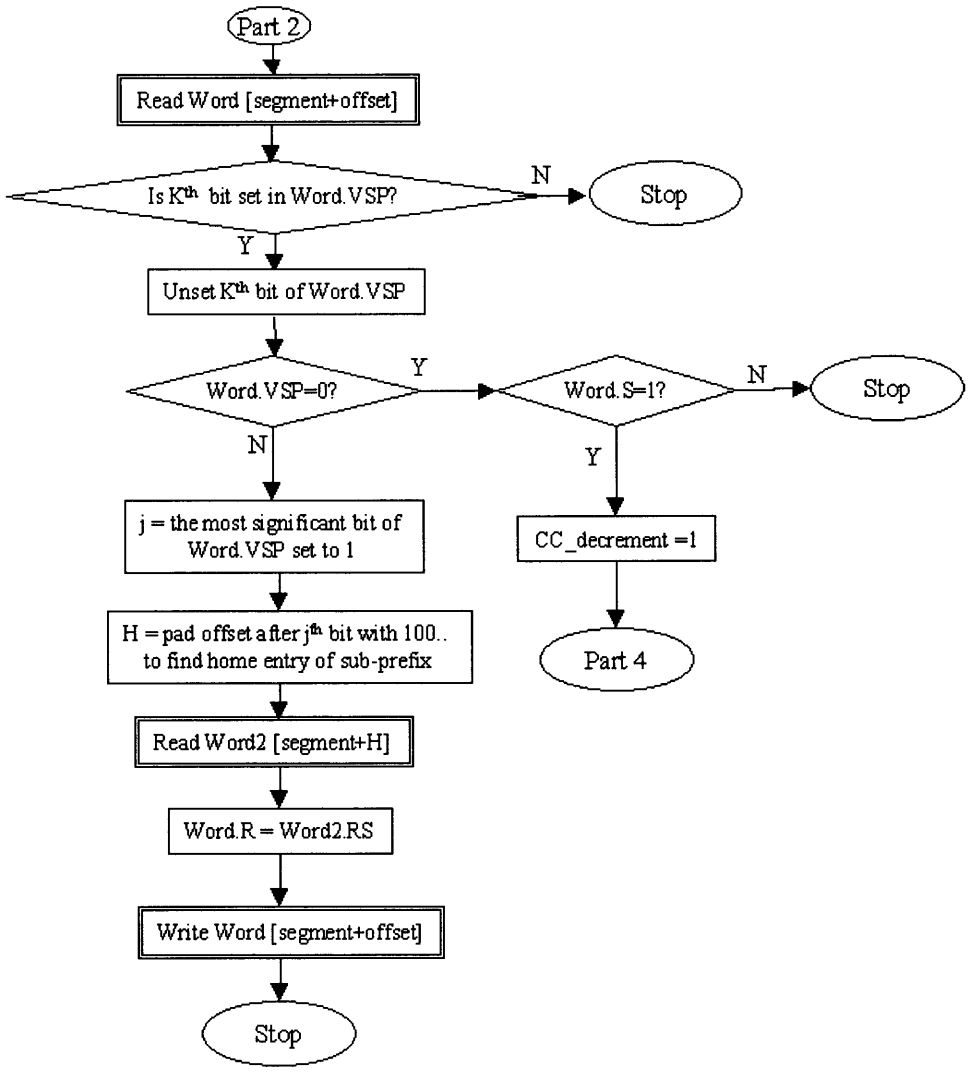

Figure B.8 Part 2 of Deletion procedure on the trie, deleting a full prefix

Deleting a full prefix requires evaluation of one entry. If the K^{th} bit of the entry has not been set then the routing information is not present and the procedure is aborted. Otherwise, K^{th} bit is unset, i.e. set to 0. If there is a valid routing information stored for this entry as signaled by VSP field, then we must find the longest prefix and its corresponding decision. We find the most significant bit of VSP set to 1 and extract the home entry corresponding to the sub-prefix. We read the home entry and insert the its routing decision stored in RS field as the new decision into the full prefix

entry. If there are no other bits in VSP set to 1 and there is no forward pointer information, i.e. S=1 then this entry has been invalidated and children counter has to be decremented by one and procedure will proceed to Part 4.

To delete a sub-prefix, we have to evaluate all affected entries. As in insertion procedure, we start evaluation from the last affected entry given by Z as offset in the node. If L^{th} bit is not set then the information is not present. Otherwise, L^{th} bit is unset. If there is any other routing information inserted previously for a shorter prefix, it is extracted from its home entry and inserted. We keep track of deleted entries and proceed to Part 4 to write children counter of the node. Part 4 write back the last affected entry of the node and updates children counter of the node as shown in Figure B.10. If the children counter of the node is zero after decreasing, then this node does not contain any routing information. It has to be deleted from the trie and appended to ILL. This procedure is given in Figure B.11.

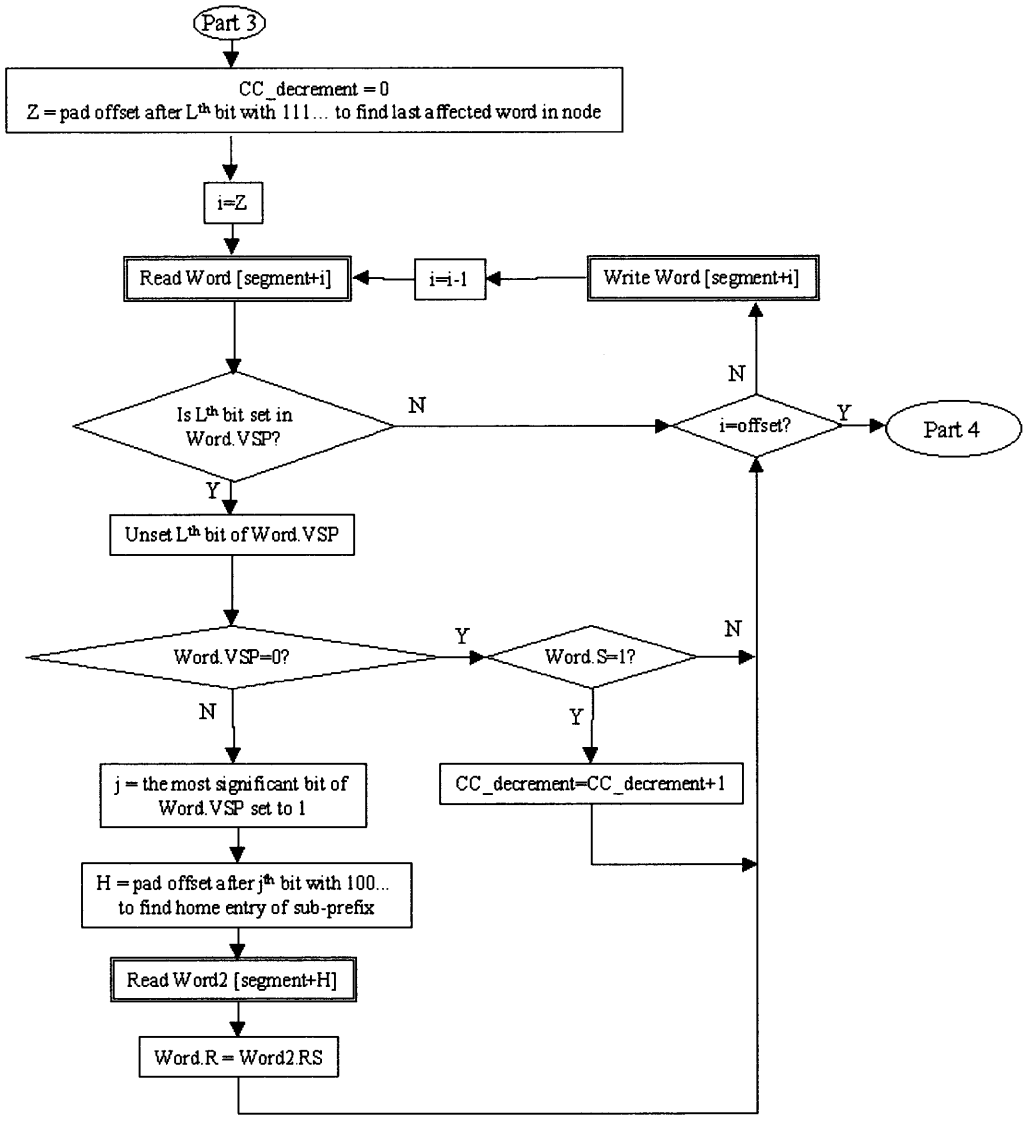

Figure B.9 Part 3 of Deletion procedure on the trie, deleting a sub-prefix

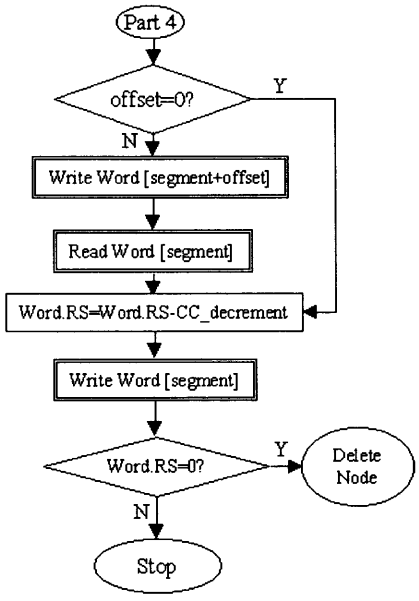

Figure B.10 Part 4 of Deletion procedure on the trie, writing children counter of the node

Deletion of a node as shown in Figure B.11 starts with adding the current node to ILL. Then it requires an evaluation of the entries that have been pushed to deletion stack during traversal of the trie at the beginning of the procedure. When an entry is popped from the stack S field is set to 1 because the node followed from its forward pointer has been deleted. If there is no other routing information as signaled by VSP field then this entry is deleted. If this entry was the only entry containing information in this node, i.e. children counter of the node was 1 then the node itself has to be deleted from the trie. Note that the entry is initialized and written back even if the node is deleted in order to maintain that an empty node is returned to ILL in its initial state. If the entry is invalidated but the node is not to be deleted then children counter is decremented only.

Adding a deleted node to ILL is straightforward. It involves one write operation to append the new node to the tail of the list given by the tail pointer (ILL_TP) which is then updated to point to the node which has just been appended.

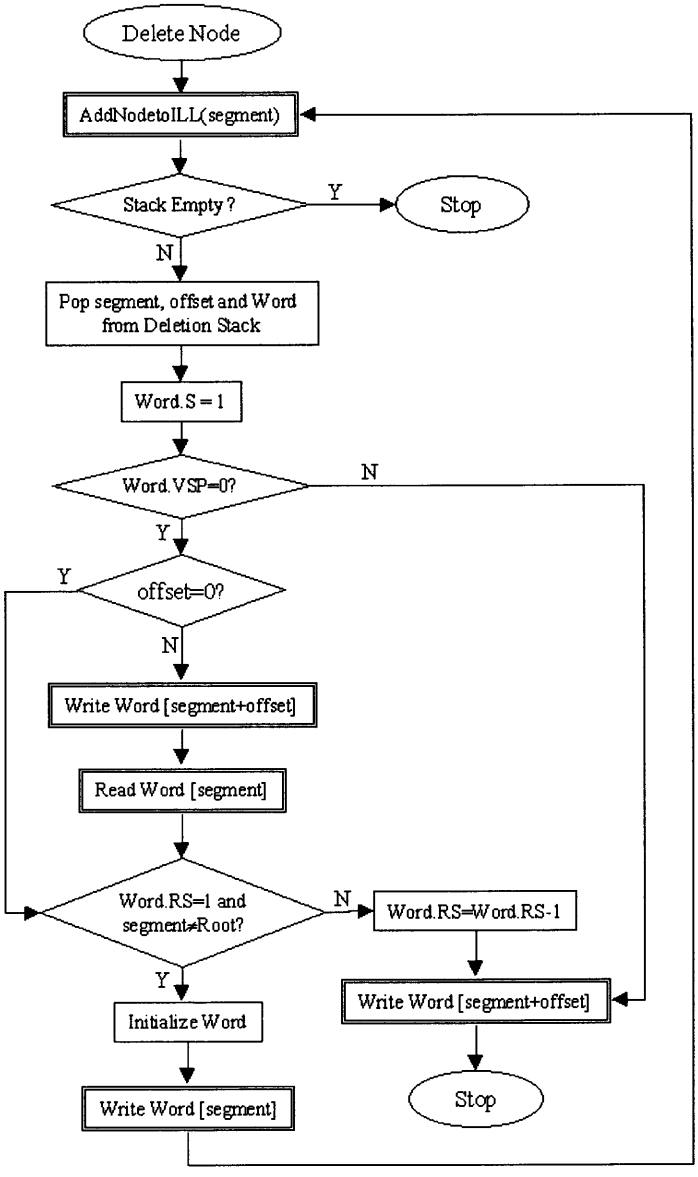

Figure B.11 Deletion of a node from the trie

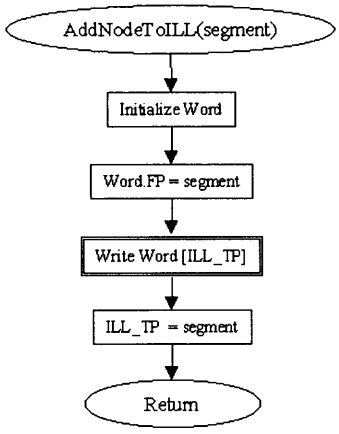

Figure B.12 Adding a deleted node to the linked list of idle nodes

APPENDIX C

LLCAT SIMULATION SOURCE CODE

In this appendix, the source code for the simulation of hardware edsign of the LLCAT algorithm is provided. The code is written in C language.

```
// L L C A T   S I M U L A T I O N
// Pınar Altın Yılmaz
// NJIT, February 99

#include <stdio.h>
#include <stdlib.h>

// Constants
//*********************************************************************
#define TRUE            1
#define FALSE           0
#define IP_BITS         32
#define ROOT_SIZE       256     // 2^8 entries in trie root

#define NUMBER_OF_NODES 25      // number of nodes in memory

char *rout_in_file = 0;
char *traf_in_file = 0;
char *upd_in_file = 0;
char *rout_out_file = 0;
char *traf_out_file = 0;
char *upd_out_file = 0;
char *trie_out_file = 0;


// Type Definitions
/*********************************************************************/
typedef unsigned char SFlag_Type;
  // 1 bit flag, stop bit
typedef unsigned VSPFlag_Type;
  // it has to be 8 or K bits wide, whichever is larger
typedef unsigned long IP_Type; // must be as wide as IP_BITS
typedef unsigned long Routing_Decision_Type;
typedef long Trie_Node_Pointer_Type;

typedef struct trie_entry {
  SFlag_Type S;
  VSPFlag_Type VSP;
  Routing_Decision_Type R,RS;
  Trie_Node_Pointer_Type FP;
}Trie_Entry_Type;
```

```
// Statistics Variables
/*********************************************************************/

long Node_Counter = 0;
long Read_Counter = 0;
long Write_Counter = 0;
long Lookup_Counter = 0;
long Insertion_Counter = 0;
long Deletion_Counter = 0;

// Globals
/*********************************************************************/

Trie_Node_Pointer_Type ILL_HP, ILL_TP;         // Idle Link List pointers
Trie_Node_Pointer_Type DS[16];                 // deletion stack
Trie_Entry_Type DSE[16];               // deletion stack
int dsp=0;
  // deletion stack pointer, points to next empty slot
Routing_Decision_Type default_rx = 16843009;   // 1.1.1.1

unsigned K=4;
unsigned NODE_SIZE;     // 2^K entries in a node
unsigned long MEMORY_SIZE;  // number of trie entries in memory

Trie_Entry_Type *Memory;

/* First 2^8 words of memory is the root of trie,
   next blocks of 2^K nodes are trie nodes
*/

// Functions
/*********************************************************************/
void setbit(VSPFlag_Type &v,int i){
// set ith least significant bit of v
v = v | (1<<(i-1));
}
/*-------------------------------------------------------------------*/
void resetbit(VSPFlag_Type &v,int i){
// reset ith least significant bit of v
v = v & ~(1<<(i-1))   ;
}
/*-------------------------------------------------------------------*/
int isbitset(VSPFlag_Type v,int i){
// is ith least significant bit of v set or not?
// returns non-zero if set
return (v & (1<<(i-1)));
}
/*-------------------------------------------------------------------*/
void initialize_entry(Trie_Entry_Type &t){
t.S=1;
```

```
t.VSP=0;
t.R=0;
t.RS=0;
t.FP=0;
}
/*-------------------------------------------------------------------*/
void initialize_memory(Trie_Entry_Type *M){
long i;
for (i=0;i<MEMORY_SIZE;i++) initialize_entry(M[i]);
ILL_HP=ROOT_SIZE;       // first node in memory after trie root
ILL_TP=MEMORY_SIZE-NODE_SIZE;
for (i=ILL_HP;i<ILL_TP;i+=NODE_SIZE) M[i].FP=i+NODE_SIZE;
}
/*-------------------------------------------------------------------*/
void read_entry(Trie_Entry_Type &t, unsigned int index){
++Read_Counter;
t.S = Memory[index].S;
t.VSP = Memory[index].VSP;
t.R = Memory[index].R;
t.RS = Memory[index].RS;
t.FP = Memory[index].FP;
}
/*-------------------------------------------------------------------*/
void write_entry(Trie_Entry_Type &t, unsigned int index){
++Write_Counter;
Memory[index].S = t.S;
Memory[index].VSP = t.VSP;
Memory[index].R = t.R;
Memory[index].RS = t.RS;
Memory[index].FP = t.FP;
}
/*-------------------------------------------------------------------*/
Trie_Node_Pointer_Type GetNodeFromILL(){

++Node_Counter;
if (Node_Counter==NUMBER_OF_NODES){
  printf("\nNo more nodes in memory, exiting...\n");
  exit(0);
}
Trie_Entry_Type TE;
Trie_Node_Pointer_Type Node=ILL_HP;
read_entry(TE,ILL_HP);
ILL_HP = TE.FP;
return Node;
}

/*-------------------------------------------------------------------*/
void AddNodeToILL(Trie_Node_Pointer_Type CN){

--Node_Counter;
Trie_Entry_Type TE;
initialize_entry(TE);
```

```
TE.FP = CN;
write_entry(TE,ILL_TP);
ILL_TP = CN;
}
/*---------------------------------------------------------------------*/
void PushDS(Trie_Node_Pointer_Type CN, Trie_Entry_Type &TE){
DS[dsp]=CN;
DSE[dsp].S = TE.S;
DSE[dsp].VSP = TE.VSP;
DSE[dsp].R = TE.R;
DSE[dsp].RS = TE.RS;
DSE[dsp].FP = TE.FP;
++dsp;
}
/*---------------------------------------------------------------------*/
int PopDS(Trie_Node_Pointer_Type &Pos, Trie_Entry_Type &TE){
if(dsp==0) return(-1);
else {
  --dsp;
  Pos = DS[dsp];
  TE.S=DSE[dsp].S;
  TE.VSP=DSE[dsp].VSP;
  TE.R=DSE[dsp].R;
  TE.RS=DSE[dsp].RS;
  TE.FP=DSE[dsp].FP;
  return(0);
}
}
/*---------------------------------------------------------------------*/
void DeleteTrieNode(){

Trie_Entry_Type TE,TE2;
Trie_Node_Pointer_Type Node, CN;

PopDS(CN,TE);
Node = NODE_SIZE * ((CN-ROOT_SIZE)/NODE_SIZE)+ROOT_SIZE;
AddNodeToILL(Node);

while ((PopDS(CN,TE))!=-1) {
  if (CN>=ROOT_SIZE)
         Node = NODE_SIZE * ((CN-ROOT_SIZE)/NODE_SIZE)+ROOT_SIZE;
  else Node = 0;
  if (CN==Node){

         if (TE.VSP==0){
                if ((TE.RS==1)&&(Node!=0)) {
                       initialize_entry(TE);
                       write_entry(TE,Node);
                       AddNodeToILL(Node);

                }
                else{
```

```
                        TE.S = 1;
                        --TE.RS;
                        write_entry(TE,Node);
                        return;
                }
        }
        else{
                TE.S =1;
                write_entry(TE,Node);
                return;
        }
 }
 else{

        if(TE.VSP==0){
                read_entry(TE2,Node);

                if ((TE2.RS==1)&&(Node!=0)) {
                        initialize_entry(TE);
                        write_entry(TE,Node);
                        write_entry(TE,CN);
                        AddNodeToILL(Node);
                }
                else{
                        --TE2.RS;
                        write_entry(TE2,Node);
                        TE.S = 1;
                        write_entry(TE,CN);
                        return;
                }
        }
        else{
                TE.S = 1;
                write_entry(TE,CN);
                return;
        }
 }
}
}
/*-------------------------------------------------------------------*/
Routing_Decision_Type search_LLCAT(IP_Type ip){

unsigned N = ((IP_BITS-8)/K) + 1; // depth of trie is N

Trie_Entry_Type TE;
Trie_Node_Pointer_Type CN=0;
unsigned i;
unsigned long offset = ip >> (IP_BITS-8); // index in root
Routing_Decision_Type longest_match = default_rx;

for (i=1;i<=N;i++){
 read_entry(TE, CN+offset);
```

```
 if(TE.S==1)
        if (TE.VSP==0) return longest_match;
        else return TE.R;
 else{
        CN = TE.FP;
        if (TE.VSP !=0) longest_match= TE.R;
 }
 offset = (ip << ((i-1)*K+8)) >> (IP_BITS-K);
}
return longest_match;
}
/*-----------------------------------------------------------------*/

Routing_Decision_Type search_LLCAT2(IP_Type ip){
// search with skipping empty nodes

unsigned N = ((IP_BITS-8)/K) + 1; // depth of trie is N

Trie_Entry_Type TE;
Trie_Node_Pointer_Type CN=0;
unsigned i;
unsigned long offset = ip >> (IP_BITS-8); // index in root
Routing_Decision_Type longest_match = default_rx;

for (i=1;i<=N;i++){
 read_entry(TE, CN+offset);

 if(TE.S==1)
        if (TE.VSP==0) return longest_match;
        else return TE.R;
 else{

        //**********
        if ( (Memory[CN].RS==1)&&(Memory[CN+offset].VSP==0) && (CN!=0)  )
                --Read_Counter;
        //*********

        CN = TE.FP;
        if (TE.VSP !=0) longest_match= TE.R;
 }
 offset = (ip << ((i-1)*K+8)) >> (IP_BITS-K);
}
return longest_match;
}
/*-----------------------------------------------------------------*/

void insert_prefix(IP_Type P,unsigned Length,Routing_Decision_Type RX){

Trie_Entry_Type TE;
unsigned i,L,M,B,H,Z;
Trie_Node_Pointer_Type CN, next_node;
int CC_increment = 0;       // increment CC by this number;
```

```
int CN_new = FALSE;
unsigned long offset;

initialize_entry(TE);

// break P into blocks of K bits
// in the least significant block, you have L bits, L<=K
// where B is the binary value of the block padded with zeros
// find the related node, CN, by traversing the trie
// insert nodes in the trie if necessary, mark CN_new=TRUE if you do

if (Length<=8){ // inserting into the root node, depth 0
// no need to keep track of CC of the root, because root is never deleted!!!
 B = P >> (IP_BITS-8);  // first 8 bits of prefix
 if (Length==8){        // inserting full prefix into root
        read_entry(TE,B);
        if (TE.VSP==0){ // entry is not valid
                setbit(TE.VSP,8);
                TE.R = RX;
                write_entry(TE,B);     // write word
                if (TE.S !=0){
                        read_entry(TE,0);  // increment CC of root
                        ++TE.RS;
                        write_entry(TE,0);
                }
        }
        else{
                setbit(TE.VSP,8);
                TE.R = RX;
                write_entry(TE,B);     // no need to increment CC
        }
 }
 else{ // inserting sub-prefix into root

     // pad block 10.. to find home entry H
     // pad block with 1's to find last affected entry Z
     H = B | (1<<(8-Length-1));
     Z = B | ((1<<(8-Length))-1);

     for (i=Z;i>=B;--i){
            read_entry(TE,i);          // read word
            if (i==H)  TE.RS = RX;
            if (TE.VSP == 0)  {   // entry is not valid
                    if (TE.S!=0) ++CC_increment;
                    TE.R = RX;
                    setbit(TE.VSP,Length);
            }
            else {       // entry is valid
                   setbit(TE.VSP,Length);
                   if ((TE.VSP>>Length)==0) TE.R = RX;
                   // no bits higher than L set to 1
            }
```

```
                write_entry(TE,i); // write word
        }
        if (CC_increment != 0){
                read_entry(TE,0);       // read CC
                TE.RS = TE.RS + CC_increment;
                write_entry(TE,0);      // write CC
        }

    }
}

else{    // inserting into depth 1 or below
// traverse root node
offset = P >> (IP_BITS-8);      // first 8 bits of prefix
read_entry(TE,offset);
if (TE.S==1){    // create new node
    CN_new = TRUE;
    TE.S = 0;
    TE.FP = GetNodeFromILL();
    write_entry(TE,offset);
    CN = TE.FP;

    if (TE.VSP==0){
            read_entry(TE,0); // increment CC of root node
            ++TE.RS;
            write_entry(TE,0);
    }
}
else CN = TE.FP;

// now traverse lower levels, to the related node, create nodes if necessary
M = (Length-8) / K;  // number of K bit blocks in prefix
if ((M!=0)&&((Length-8)%K==0)) --M;
L = Length-M*K-8;       // remaining bits in least significant block of prefix

for (i=1;i<=M;i++){

    // offset = ith block of K bits in P;
    offset = (P << ((i-1)*K+8)) >> (IP_BITS-K);

    if (CN_new==FALSE){ // currently in an old node
            read_entry(TE,CN+offset);
            if (TE.S==0){
                    CN =TE.FP;
                    continue;
            }
            CN_new  = TRUE; // create a new node
            TE.S = 0;
            next_node = TE.FP = GetNodeFromILL();
            if (offset== 0){
                    if (TE.VSP==0) TE.RS = TE.RS + 1;
                    write_entry(TE,CN);     // write word and CC
```

```
                CN = next_node;
        }
        else{
                write_entry(TE,CN+offset);          // write word
                if (TE.VSP==0){
                        read_entry(TE,CN); // increment CC of node
                        ++TE.RS;
                        write_entry(TE,CN);
                }
                CN = next_node;
        }
 }
 else{ // currently in a new node
        // no need to read anything, create another new node
        // configure forward pointer and CC
        initialize_entry(TE);
        TE.S = 0;
        next_node = TE.FP = GetNodeFromILL();
        if (offset== 0){
                TE.RS = 1;
                write_entry(TE,CN);    // write word and CC
                CN = next_node;
        }
        else{
                write_entry(TE,CN+offset);          // write word
                initialize_entry(TE);
                TE.RS = 1;
                write_entry(TE,CN);    // write CC
                CN = next_node;
        }
       }
} // for

B = (P << (  M*K+8  )    ) >> (IP_BITS-K);      // last block of P

if (L==K)     // P is a full prefix

    if (CN_new == TRUE){       // inserting full prefix into a new node
      initialize_entry(TE);
      TE.R = RX;
      setbit(TE.VSP,K);
      if (B==0) { // first entry in the block
             TE.RS = 1;
             write_entry(TE,CN);   // write word and CC
      }
      else{
             write_entry(TE,CN+B);   // write word
             initialize_entry(TE);
             TE.RS = 1;
             write_entry(TE,CN);  // write CC
      }
    }
```

```
    else{        // inserting full prefix into an old node
      read_entry(TE,CN+B);          // read word
      if (TE.VSP == 0){    // entry is not valid
              initialize_entry(TE);
              setbit(TE.VSP,K);
              TE.R = RX;
              if (B==0){
                  TE.RS = TE.RS + 1;
                  write_entry(TE,CN);  //write word and CC
              }
              else{
                  write_entry(TE,CN+B);  // write word
                  read_entry(TE,CN);  // read CC
                  TE.RS = TE.RS + 1;
                  write_entry(TE,CN);  // write CC
              }
      }
      else{        // entry is valid
              setbit(TE.VSP,K);
              TE.R = RX;
              write_entry(TE,CN+B);          //write entry
      }
    }

else            // P is a sub-prefix
    if (CN_new == TRUE){       // inserting sub-prefix into a new node
     // pad block 10.. to find home entry H
     // pad block with 1's to find last affected entry Z
     H = B | (1<<(K-L-1));
     Z = B | ((1<<(K-L))-1);

     for (i=Z;i>B;--i){
                 initialize_entry(TE);
                 if (i==H)  TE.RS = RX;
                 setbit(TE.VSP,L);
                 ++CC_increment;
                 TE.R = RX;
                 write_entry(TE,CN+i); // write word
     }

     initialize_entry(TE);
     setbit(TE.VSP,L);
     ++CC_increment;
     TE.R = RX;
     if (B==0) {
              TE.RS = CC_increment;
              write_entry(TE,CN);  // write word and CC
     }
     else{
              write_entry(TE,CN+B); // write word
              initialize_entry(TE);
              TE.RS = CC_increment;
```

```
            write_entry(TE,CN);  // write CC
        }
        }
        else{        // inserting sub-prefix into an old node

          // pad block 10.. to find home entry H
          // pad block with 1's to find last affected entry Z
          H = B | (1<<(K-L-1));
          Z = B | ((1<<(K-L))-1);

          for (i=Z;i>B;--i){
                  read_entry(TE,CN+i);          // read word
                  if (i==H)  TE.RS = RX;
                  if (TE.VSP == 0)  {   // entry is not valid
                          ++CC_increment;
                          TE.R = RX;
                          setbit(TE.VSP,L);
                  }
                  else {        // entry is valid
                         setbit(TE.VSP,L);
                         if ((TE.VSP>>L)==0) TE.R = RX;
                         // no bits higher than L set to 1
                  }
                  write_entry(TE,CN+i); // write word
          }

          read_entry(TE,CN+B);          // read word
          if (TE.VSP == 0)  {   // entry is not valid
               setbit(TE.VSP,L);
               ++CC_increment;
               TE.R = RX;
          }
          else{        // entry is valid
                 setbit(TE.VSP,L);
                 if ((TE.VSP>>L)==0 ) TE.R = RX;
                 // (no bits higher than L set to 1)
          }

          if (CC_increment == 0)  write_entry(TE,CN+B);
          else
              if (B==0) {
                      TE.RS = TE.RS + CC_increment;
                      write_entry(TE,CN);
              }
              else{
                      write_entry(TE,CN+B);      // write word
                      read_entry(TE,CN);       // read CC
                      TE.RS = TE.RS + CC_increment;
                      write_entry(TE,CN);       // write CC
              }
    }
} // else, not in root
```

```
} // function

/*-------------------------------------------------------------------*/

void delete_prefix(IP_Type P, unsigned Length){

Trie_Entry_Type TE,TE2;
unsigned i,j,q,r,L,M,B,H,Z;
Trie_Node_Pointer_Type CN;
int CC_decrement = 0;       // decrement CC by this number;
unsigned long offset;

dsp = 0;
CN = 0;
initialize_entry(TE);       // a trie entry where all bits but S is 0.

// break P into blocks of K bits
// in the least significant block, you have L bits, L<=K
// where B is the binary value of the block padded with zeros
// find the related node, CN, by traversing the trie
// insert nodes in the trie if necessary, mark CN_new=TRUE if you do

M = (Length-8) / K;  // number of K bit blocks in prefix
if ((M!=0)&&((Length-8)%K==0)) --M;
L = Length-M*K-8;      // remaining bits in least significant block of prefix

// traverse the trie to the related node, push the nodes to the deletion stack

if (Length<=8){  // deleting a prefix from root node
  B = P  >> (IP_BITS-8);      // last block of P

  if (Length==8){       // deleting a full-prefix from root
        read_entry(TE,B);
        if (isbitset(TE.VSP,8)==FALSE)  return;
        // prefix is not inserted
        resetbit(TE.VSP,8);
        // full prefix is here to be deleted
        if (TE.VSP == 0) {      // no sub-prefixes inserted
              write_entry(TE,B);     // write word
              read_entry(TE,0);      // read CC
              --TE.RS;
              write_entry(TE,0);    // write CC
        }
        else{   // there are sub-prefixes inserted
              // find the longest prefix
              // insert its routing decision
              // no entry or node is deleted
              for(i=TE.VSP,j=0;i!=0;i=i>>1,j++);
              // jth bit of VSP is the highest bit set to 1
              // find home entry for the prefix j bits long
              H = ((B >> (8-j))<< (8-j)) | (1<<(8-j-1));
              // home entry
```

```
                /
                    read_entry(TE2,CN+H);     // read word for home entry
                    TE.R = TE2.RS;
                    write_entry(TE,CN+B);      // write word
            }
    }
    else{   // deleting a sub-prefix from root

    Z = B | ((1<<(8-Length))-1);

    for (i=Z;i>=B;--i){

            read_entry(TE,i);              // read word
            if (TE.VSP != 0) {     // entry is valid
                    resetbit(TE.VSP,Length);
                    if (TE.VSP ==0)     // no more routing info is present
                            ++CC_decrement;        // the entry has been deleted
                    else  if (isbitset(TE.VSP,K)==FALSE){
                        // no full prefix is present
                        // find longest prefix
                        // insert its routing decision
                        // no entry or node is deleted

                        for(q=TE.VSP,r=0;q!=0;q=q>>1,r++);
                        // rth bit of VSP is the highest bit set to 1
                        // find home entry for the prefix r bits long

                        H = ((B >> (8-r))<< (8-r)) | (1<<(8-r-1));
                        // home entry
                        read_entry(TE2,H); // read word for home entry
                        TE.R = TE2.RS;
                    }
            }
            write_entry(TE,i);      // write word
          }  // for
          if (CC_decrement != 0){
            read_entry(TE,0);       // read CC
            TE.RS = TE.RS - CC_decrement;
            write_entry(TE,0);       // write CC
          }
          }

}
else{     // deleting a prefix from depth 1 or below

// traverse root, push all referring entries to stack
offset = P >> (IP_BITS-8);
read_entry(TE,offset);
if (TE.S==1) return; // entry does not exist
CN = TE.FP;
PushDS(offset,TE);

// traverse to lower levels
```

```
for (i=1;i<=M;i++){
  // offset = ith block of K bits in P;
  offset = (P << ((i-1)*K+8)) >> (IP_BITS-K);
  read_entry(TE,CN+offset);
  if (TE.S==1) return; // entry does not exist
  PushDS(CN+offset,TE);     // push to deletion stack
  CN = TE.FP;
    }
PushDS(CN,TE);

B = (P << (M*K+8) ) >> (IP_BITS-K);      // last block of P

if (L==K) {     // P is a full prefix

  read_entry(TE,CN+B);
  if (isbitset(TE.VSP,K)==FALSE)  return; // prefix is not inserted
  resetbit(TE.VSP,K);
  // full prefix is here to be deleted
  if (TE.VSP == 0) {      // no sub-prefixes inserted
         if (B==0) {
                if (TE.RS == 1){     // this trie node must be deleted
                       initialize_entry(TE);
                       write_entry(TE,CN);
                       DeleteTrieNode();
                }
                else{
                       --TE.RS;
                       write_entry(TE,CN);      // write CC and word
                }
         }
         else{   // this is not the first node
                read_entry(TE2,CN);      // read CC
                if (TE2.RS == 1){ // this trie node must be deleted
                       initialize_entry(TE);
                       write_entry(TE,CN);
                       write_entry(TE,CN+B);
                       DeleteTrieNode();
                }
                else{
                       --TE2.RS;
                       write_entry(TE2,CN);    // write CC
                       write_entry(TE,CN+B);     // write word
                    }
             }

     }
  else{   // there are sub-prefixes inserted
         // find the longest prefix and insert its routing decision
         // no entry or node is deleted
         for(i=TE.VSP,j=0;i!=0;i=i>>1,j++);
         // jth bit of VSP is the highest bit set to 1
         // find home entry for the prefix j bits long
```

```
        H = ((B >> (K-j))<< (K-j)) | (1<<(K-j-1));      // home entry
        read_entry(TE2,CN+H);    // read word for home entry
        TE.R = TE2.RS;
        write_entry(TE,CN+B);     // write word
      }
    }
else{            // P is a sub-prefix

  Z = B | ((1<<(K-L))-1);
  CC_decrement = 0;

  for (i=Z;i>B;--i){

        read_entry(TE,CN+i);             // read word
        if (TE.VSP != 0) {     // entry is valid
               resetbit(TE.VSP,L);
               if (TE.VSP ==0)    // no more routing info is present
                      ++CC_decrement;        // the entry has been deleted
               else  if (isbitset(TE.VSP,K)==FALSE){
                   // no full prefix is present
                   // find longest prefix
                   // insert its routing decision
                   // no entry or node is deleted

                   for(q=TE.VSP,r=0;q!=0;q=q>>1,r++);
                   // rth bit of VSP is the highest bit set to 1
                   // find home entry for the prefix r bits long

                   H = ((B >> (K-r))<< (K-r)) | (1<<(K-r-1));
                   // home entry
                   read_entry(TE2,CN+H); // read word for home entry
                   TE.R = TE2.RS;
               }
        }
        write_entry(TE,CN+i);     // write word
      }  // for

  read_entry(TE,CN+B);
  if (TE.VSP != 0) {     // entry is valid
        resetbit(TE.VSP,L);
        if (TE.VSP ==0)    // no more routing info is present
               ++CC_decrement;        // the entry has been deleted
        else if (isbitset(TE.VSP,K)==FALSE){
               // no full prefix is present
               // find longest prefix and insert its routing decision
               // no entry or node is deleted

               for(q=TE.VSP,r=0;q!=0;q=q>>1,r++);
               // rth bit of VSP is the highest bit set to 1

               // find home entry for the prefix r bits long
               H = ((B >> (K-r))<< (K-r)) | (1<<(K-r-1));
```

```
                // home entry
                read_entry(TE2,CN+H);   // read word for home entry
                TE.R = TE2.RS;
        }
 }
 if (B==0) {
        TE.RS = TE.RS - CC_decrement;
        write_entry(TE,CN);
        if (TE.RS == 0) DeleteTrieNode();
 }
 else{
        read_entry(TE2,CN);      // read CC
        TE2.RS = TE2.RS - CC_decrement;
        if (TE2.RS == 0){  // this trie node must be deleted
                initialize_entry(TE);
                write_entry(TE,CN);
                write_entry(TE,CN+B);
                DeleteTrieNode();
        }
        else{
                write_entry(TE2,CN);        // write CC
                write_entry(TE,CN+B); // write word
        }
 }
}
}
}
/*--------------------------------------------------------------------*/

void print_trie(char *out_file){

unsigned long i,j;
Routing_Decision_Type rx;
unsigned n1,n2,n3,n4;

FILE *fp_out = fopen(out_file,"w");
if (fp_out == NULL ) { printf("cannot create trie output file..."); exit(0); }

fprintf(fp_out,"K                  : %d\n",K);
fprintf(fp_out,"Number of Nodes    : %d\n",Node_Counter);
fprintf(fp_out,"Number of Entries  : %d\n",(Node_Counter*NODE_SIZE+256));
fprintf(fp_out,"Memory Size        : %d\n\n",MEMORY_SIZE);
fprintf(fp_out,"ILL Head Pointer   : %d\n",ILL_HP);
fprintf(fp_out,"ILL Tail Pointer   : %d\n\n",ILL_TP);

/* Print out all entries, takes a lot of space */

for (i=0;i<MEMORY_SIZE;i++){

// print just the root
if (i==ROOT_SIZE) break;
```

```
if ((i==ROOT_SIZE)|| (   (i>ROOT_SIZE) && (  ((i-ROOT_SIZE)%NODE_SIZE)==0 ))  )
        fprintf(fp_out,"\n");

  fprintf(fp_out,"%d\t",i);

  for (j=16;j>0;j--){
        if(j==8) fprintf(fp_out," ");
        if (isbitset(i,j)!=FALSE) fprintf(fp_out,"1");
                else fprintf(fp_out,"0");}

  fprintf(fp_out,"\t");

  if (Memory[i].S==0) fprintf(fp_out,"0\t");
  else fprintf(fp_out,"1\t");

  for (j=8;j>0;j--)
  if (isbitset(Memory[i].VSP,j)!=FALSE) fprintf(fp_out,"1");
        else fprintf(fp_out,"0");

  fprintf(fp_out,"\t");

  rx = Memory[i].R;
  n1 = rx >> 24;
  n2 = (rx << 8) >> 24;
  n3 = (rx << 16) >> 24;
  n4 = (rx << 24) >> 24;
  fprintf(fp_out,"%u.%u.%u.%u\t",n1,n2,n3,n4);

  rx = Memory[i].RS;
  if ((i>ROOT_SIZE) && ( ((i-ROOT_SIZE)%NODE_SIZE)==0 )||(i==0))
        fprintf(fp_out,"    %u\n",rx);
  else{
        n1 = rx >> 24;
        n2 = (rx << 8) >> 24;
        n3 = (rx << 16) >> 24;
        n4 = (rx << 24) >> 24;
        fprintf(fp_out,"%u.%u.%u.%u\t",n1,n2,n3,n4);
  }

  fprintf(fp_out,"%u\n",Memory[i].FP);

}
fclose(fp_out);
}
/*-----------------------------------------------------------------*/

void read_routing_table_file(char *in_file, char *out_file){

/*
format of the file is
n.n.n.n n n.n.n.n
prefix length nexthop
```

```
        */
        unsigned s1,s2,s3,s4,n1,n2,n3,n4,length;
        IP_Type ip;
        Routing_Decision_Type rx;
        int k=1;

        FILE *fp_in = fopen(in_file,"r");
        FILE *fp_out = fopen(out_file,"w");

        if (fp_in == NULL ) { printf("cannot open routing input file...");   exit(0);}
        if (fp_out == NULL ) { printf("cannot create routing output file..."); exit(0);
}

        // fprintf(fp_out,"IP\t\tLength\tRout.Dec.\t\tReads\tWrites\n");
        fprintf(fp_out,"Length\tReads\tWrites\n");

        while (fscanf(fp_in,"%u.%u.%u.%u%u%u.%u.%u.%u\n",
          &s1,&s2,&s3,&s4,&length,&n1,&n2,&n3,&n4 )!=EOF){

          Read_Counter=0;
          Write_Counter=0;
          ip = s1;
          ip = ip << 8;
          ip = ip+s2;
          ip = ip << 8;
          ip = ip+s3;
          ip = ip << 8;
          ip = ip+s4;
          rx = n1;
          rx = rx<< 8;
          rx = rx+n2;
          rx = rx << 8;
          rx = rx+n3;
          rx = rx << 8;
          rx = rx+n4;

          insert_prefix(ip,length,rx);

          // fprintf(fp_out,"%u.%u.%u.%u\t\t%u\t%u.%u.%u.%u\t\t",
          //      s1,s2,s3,s4,length,n1,n2,n3,n4);

          fprintf(fp_out,"%u\t", length);
          fprintf(fp_out,"%u\t", Read_Counter);
          fprintf(fp_out,"%u\n", Write_Counter);
        }
        fclose(fp_in);
        fclose(fp_out);
        }
        /*--------------------------------------------------------------------*/

        void lookup_traffic(char *in_file, char *out_file){
```

```
        /*
        format of the file is
        n.n.n.n n
        IP length
        */
        unsigned s1,s2,s3,s4,n1,n2,n3,n4,pl;
        IP_Type ip;
        Routing_Decision_Type rx;
        int k=1;
        // read lookups from a file and perform lookup on trie

        FILE *fp_in = fopen(in_file,"r");
        FILE *fp_out = fopen(out_file,"w");

        if (fp_in == NULL ) { printf("cannot open traffic input file..."); exit(0); }
        if (fp_out == NULL ) { printf("cannot create traffic output file..."); exit(0);
}

        printf("\nNow printing traffic output...\n\n");

        fprintf(fp_out,"IP\t\tRout.Dec.\t\t\tReads\n");

        while (fscanf(fp_in,"%u.%u.%u.%u %u\n",&s1,&s2,&s3,&s4,&pl )!=EOF){
          Read_Counter = 0;
          ip = s1;
          ip = ip << 8;
          ip = ip+s2;
          ip = ip << 8;
          ip = ip+s3;
          ip = ip << 8;
          ip = ip+s4;

          rx = search_LLCAT(ip);
          n1 = rx >> 24;
          n2 = (rx << 8) >> 24;
          n3 = (rx << 16) >> 24;
          n4 = (rx << 24) >> 24;

          fprintf(fp_out,"%u.%u.%u.%u\t%u.%u.%u.%u\t%u\n",
                  s1,s2,s3,s4,n1,n2,n3,n4,Read_Counter);
        }
        fclose(fp_in);
        fclose(fp_out);
        }
        /*-----------------------------------------------------------------*/

        void update_trie(char *in_file, char *out_file){

        /*
        format of the file is
        D/I n.n.n.n n
        Operation IP length
```

```
*/
// read updates from a file and perform lookup on trie

FILE *fp_in = fopen(in_file,"r");
FILE *fp_out = fopen(out_file,"w");

if (fp_in == NULL ) { printf("cannot open update input file..."); exit(0); }
if (fp_out == NULL ) { printf("cannot create update output file..."); exit(0); }

unsigned s1,s2,s3,s4,n1,n2,n3,n4,length;
IP_Type ip;
Routing_Decision_Type rx;
char op;

printf("\nNow printing update output...\n\n");

fprintf(fp_out,"Op.Type\tLength\tReads\tWrites\tTotal\n");

while (fscanf(fp_in,"%1s%u.%u.%u.%u%u",&op,&s1,&s2,&s3,&s4,&length )!=EOF){

  Read_Counter = 0;
  Write_Counter = 0;

  ip = s1;
  ip = ip << 8;
  ip = ip+s2;
  ip = ip << 8;
  ip = ip+s3;
  ip = ip << 8;
  ip = ip+s4;

  if (op=='D'){  // delete prefix
         fscanf(fp_in,"\n");
         delete_prefix(ip,length);
  }
  else {  // insert prefix
         fscanf(fp_in,"%u.%u.%u.%u\n", &n1,&n2,&n3,&n4);
         rx = n1;
         rx = rx<< 8;
         rx = rx+n2;
         rx = rx << 8;
         rx = rx+n3;
         rx = rx << 8;
         rx = rx+n4;
         insert_prefix(ip,length,rx);
  }

  // standard output for diagnostics
  printf("%c\t%u.%u.%u.%u\t ", op, s1,s2,s3,s4);
  printf("%u\t",length);
  printf("%d\t",Read_Counter);
  printf("%d\t",Write_Counter);
```

```
 printf("%d\n",Read_Counter+Write_Counter);

 if (op=='D') printf("\n");
 else printf("\t%u.%u.%u.%u\n",n1,n2,n3,n4);

 fprintf(fp_out, "%c\t%u.%u.%u.%u\t ", op, s1,s2,s3,s4);
 fprintf(fp_out, "%u\t",length);
 fprintf(fp_out, "%d\t",Read_Counter);
 fprintf(fp_out, "%d\t",Write_Counter);
 fprintf(fp_out, "%d\n",Read_Counter+Write_Counter);
}

fclose(fp_in);
fclose(fp_out);
}
/*-----------------------------------------------------------------*/
void find_chain(int start, int node, int level){
// count all nodes in the chain except the root
// effectively print nodes in chain

unsigned i, new_node;

if ((level) && (Memory[node].RS==1) && (Memory[start].VSP!=0) )
 printf("#%d\n", level);

if (Memory[node].RS==1){
 if (node) level++;
}
else level=0;

if (Memory[start].S==0){
 // go down the trie
 if (node && level) printf("%d\t",start-node);
 new_node = Memory[start].FP;
 if (level && (Memory[new_node].RS!=1))
        printf("#%d\n", level);

 for (i=0;i<NODE_SIZE;i++){
        find_chain(new_node+i, new_node, level);
 }
}
}
/*-----------------------------------------------------------------*/
void find_chain2(int start, int node, int level){
// to count empty nodes where CC=1 due to FP

unsigned i, new_node;

if ((Memory[node].RS==1)&&(Memory[start].S==0)&&(Memory[start].VSP==0))
   printf("Node: %d      %d\n",node, level);

if (Memory[start].S==0){
```

```
  // go down the trie
  level++;
  new_node = Memory[start].FP;
  for (i=0;i<NODE_SIZE;i++){
        find_chain2(new_node+i, new_node, level);
  }
}
}
/*********************************************************************/
int main(int argc, char *argv[]){

K = *(argv[1])-'0';
rout_in_file = argv[2];
rout_out_file = argv[3];
traf_in_file = argv[4];
traf_out_file = argv[5];
upd_in_file = argv[6];
upd_out_file = argv[7];
trie_out_file = argv[8];

NODE_SIZE = (1<<K);     // 2^K entries in a node
MEMORY_SIZE = NUMBER_OF_NODES*NODE_SIZE + ROOT_SIZE;
// number of trie entries in memory
Memory = (Trie_Entry_Type *) malloc(sizeof(Trie_Entry_Type)*MEMORY_SIZE);

initialize_memory(Memory);
read_routing_table_file(rout_in_file, rout_out_file);
lookup_traffic(traf_in_file, traf_out_file);
update_trie(upd_in_file, upd_out_file);
print_trie(trie_out_file);

/*
printf("\n");
printf("K          : %d\n",K);
printf("Number of Nodes    : %d\n",Node_Counter);
printf("Number of Entries  : %d\n",(Node_Counter*NODE_SIZE+256));
printf("Memory Size        : %d\n\n",MEMORY_SIZE);
printf("ILL Head Pointer   : %d\n",ILL_HP);
printf("ILL Tail Pointer   : %d\n\n",ILL_TP);

int i;
for (i=0;i<ROOT_SIZE;i++){
  find_chain(i,0,0);
}
*/
free(Memory);
return 0;
}
/*********************************************************************/
```

REFERENCES

1. A. Belenkiy and N. Uzun, "Deterministic IP Table Look-Up at Wire Speed," INET'99: The Internet Global Summit, San Jose, CA, June 1999.
2. T. Chiueh and P. Pradhan, "Cache Memory Design for Internet Processors," Proceedings of IEEE Hot Interconnects-VII, August 18-20,1999.
3. M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," presented at the SIGCOMM'97, Cannes, France.
4. W. Doeringer, G. Karjoth, and M. Nassehi, "Routing on longest-matching prefixes," IEEE/ACM Trans. Networking, vol. 4, pp. 86-97, Feb. 1996.
5. V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy", RFC1519.
6. "Funet Routing Table and Traces," http://www.nada.kth.se/~snilsson/public/code/router/, September 1999.
7. P. Gupta, N. McKeown, and S. Lin, "Routing lookups in hardware at memory access speeds," in Proc. IEEE INFOCOM'98 Conf., pp. 1240-1247.
8. Internet Software Consortium, "Internet Domain Survey, July 1999" http://www.isc.org/dsview.cgi?domainsurvey/WWW-9907/report.html
9. C. Labovitz, G. Malan, and F. Jahanian, "Internet routing instability," IEEE Trans. Networking, Vol. 6 Number 5, Page 515, Oct. 1998.
10. B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," IEEE Trans. Networking, Vol. 7 Number 3, June 1999, Page 324.
11. A. McAuley and P. Francis, "Fast routing table lookup using CAMs," INFOCOM'93, Page 1382-1391, March-April 1993.
12. Merit Network, Inc., "Internet Performance Measurement and Analysis Project" http://www.merit.edu/IPMA, September 1999.
13. S. Nilsson and G. Karlsson, "Fast address look-up for Internet routers," in Proc. IEEE Broadband Communications'98, Stuttgart, Germany.
14. S. Nilsson and G. Karlsson,"IP-Address Lookup Using LC-Tries", IEEE Journal on Selected Areas in Communications, Volume 17 Number 6, June 1999, Page 1083.
15. K. Sklower, "A tree-based routing table for Berkeley Unix," presented at the 1991 Winter Usenix Conf., Dallas, TX.
16. V. Srinivasan and G. Varghese, "Fast IP lookups using controlled prefix expansion," ACM TOCS, vol. 17, pp. 1-40, Feb. 1999.
17. H.Y. Tzeng, T. Przygienda, "On Fast Address-Lookup Algorithms", IEEE Journal on Selected Areas in Communications, Volume 17 Number 6, June 1999, Page 1067.
18. "VBNS Route Information", http://www.vbns.net/route/Allsnap.rt.html, September 1999.
19. M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in Proc. ACM SIGCOMM'97 Conf., Cannes, France, pp. 25-35.