IP ROUTING TABLE COMPRESSION USING TCAM AND DISTANCE-ONE MERGE

A Thesis

by

KALYANA CHAKRAVARTHY BOLLAPALLI

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

MASTER OF SCIENCE

December 2009

Major Subject: Computer Engineering

IP ROUTING TABLE COMPRESSION USING TCAM AND DISTANCE-ONE MERGE

A Thesis

by

KALYANA CHAKRAVARTHY BOLLAPALLI

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

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Sunil P. Khatri
Committee Members,	Weiping Shi
	Hank Walker
Head of Department,	Costas N. Georghiades

December 2009

Major Subject: Computer Engineering

ABSTRACT

IP Routing Table Compression Using TCAM and Distance-One Merge.

(December 2009)

Kalyana Chakravarthy Bollapalli, B. Tech., Indian Institute of Technology, Bombay

Chair of Advisory Committee: Dr. Sunil P. Khatri

In an attempt to slow the exhaustion of the Internet Protocol (IP) address space, Class-less Inter-Domain Routing (CIDR) was proposed and adopted. However, the decision to utilize CIDR also increases the size of the routing table, since it allows an arbitrary partitioning of the routing space. We propose a scheme to reduce the size of routing table in the CIDR context. Our approach utilizes a well-known and highly efficient heuristic to perform 2-level logic minimization in order to compress the routing table. By considering the IP routing table as a set of completely specified logic functions, we demonstrate that our technique can achieve about 25% reduction in the size of IP routing tables, while ensuring that our approach can handle routing table updates in real-time. The resulting routing table can be used with existing routers without needing any change in architecture. However, by realizing the IP routing table as proposed in this thesis, the implementation requires less complex hardware than Ternary CAM (TCAM) which are traditionally used to implement IP routing tables. The proposed architecture also reduces lookup latency by about 46%, hardware area by 9% and power consumed by 15% in contrast to a TCAM based implementation.

To my family and friends

ACKNOWLEDGMENTS

I would like to thank my adviser, Dr. Sunil P. Khatri, for his guidance, encouragement and support during the course of my masters program. I would also like to thank my research group members for their valuable suggestions in various aspects of my research.

I would like to thank my family for their moral support when I needed it. A final note of thanks to Texas A&M University and the Department of Electrical and Computer Engineering for giving me the opportunity to pursue my master's degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

IP addresses were originally partitioned using a *Class-based scheme*. The class of a network can be identified by specific ranges of IP addresses that belong to the network. The range of addresses spanned by a network is specified by a prefix length. Each IP address (IPv4) can be visualized as a 32-bit number and the prefix length specifies the number of most significant bits that are common to all IP addresses (prefix) in the network. In the Class-based scheme networks were divided into 3 classes, class A, class B and class C. Class A, B or C networks utilized 8, 16 and 24 bit prefix lengths respectively. As the number of computers increased, available IP addresses decreased and the need for available IP addresses increased. This led to a compelling need to utilize IP addresses more efficiently. This motivated the introduction of Class-less Inter-Domain Routing (CIDR) [1, 2] in 1993. In this method, networks were permitted to have an arbitrary number of IP addresses, allowing a more flexible IP allocation. In the CIDR approach, the IP addresses allocated to a network are represented as a list of IP address ranges. Each range of IP addresses is represented as a prefix and a prefix length. Unlike the Class-based scheme, the prefix lengths are now arbitrary integer values from 0 to 32. The downside of this choice was that it results in an increase in the size of IP routing tables, due to the fine granularity of IP address allocation.

This thesis follows the style of *IEEE Transactions on Networking*.

The fine grained allocation of IP addresses and the rapid growth in the world-wide networking infrastructure in the late 1990's fueled a super-linear growth in global routing tables. This super-linear growth continued until late 2001, threatening an eventual widespread breakdown of connectivity. In an attempt to prevent this from happening, there was a cooperative effort by internet service providers (ISPs) to keep the global routing table as small as possible, by using CIDR with route aggregation. With route aggregation blocks of contiguous IP address ranges of same are merged into a single range. While this slowed the growth of the routing table to a linear process for a few years, with the expanded demand for multihoming by end-user networks, the growth was once again exponential by the middle of 2004. The global routing table hit 200,000 entries in October 2006 [3] and 300,000 entries in August 2009 [4]. The growth in the global IP routing table size is plotted in Figure I.1. This growth in routing table requires a commensurate growth in hardware resources, thus rendering even the more expensive routers obsolete sooner. Thus there is a dire necessity to reduce the size and the rate of growth of the routing table. In this thesiswe address both these concerns.

When a packet is to be routed from its originating point to its destination, every router encountered in the path of the packet performs 2 routing lookups. One to verify that a path from source to router exists, and the other to identify the next hop (next router or destination). Routing lookup involves performing a comparison of the packet's IP address against every routing table entry (also referred to as a route). An increase in the size of routing table increases the time taken to identify the next hop. To reduce the time taken to identify the next hop, many algorithms have been proposed [5, 6, 7, 8, 9, 10, 11, 12]. Algorithmic approaches implemented in software suffice for routers that handle low data rates. However, they are overwhelmed by the large data rates that are in excess of 40 Gbps. This shortcoming can be

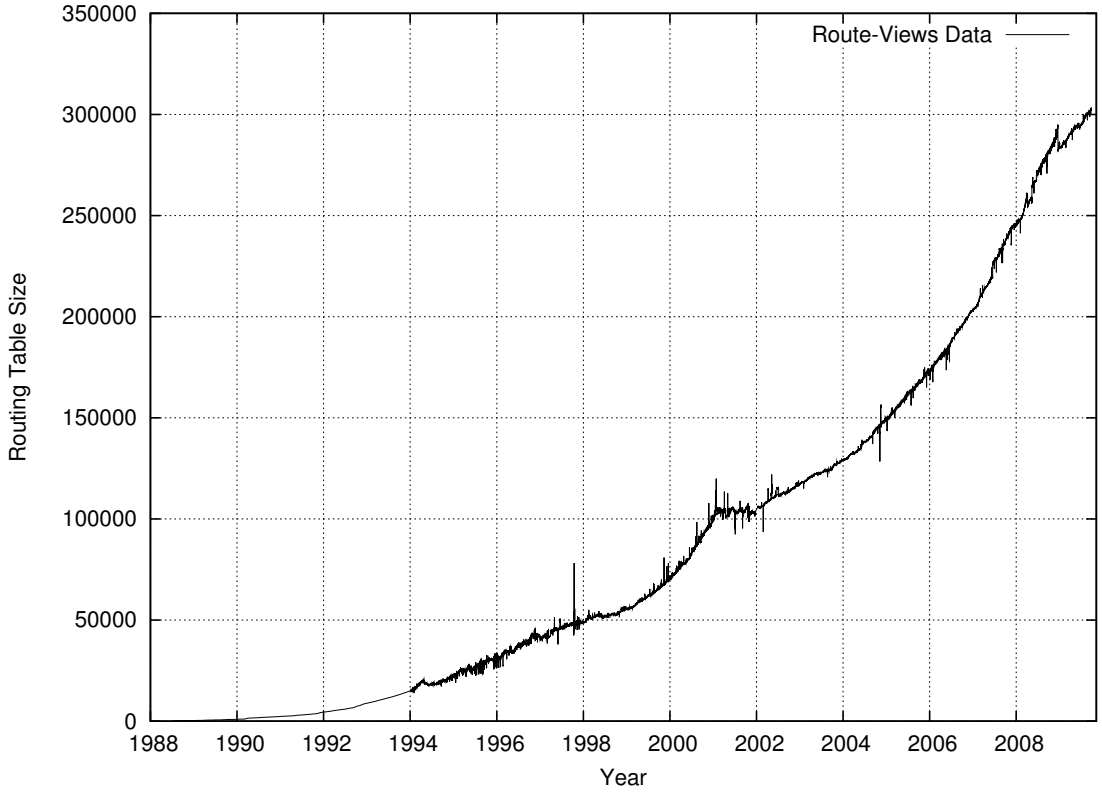

Fig. I.1. Global Routing Table Growth

attributed to the multiple memory accesses performed per packet by the algorithmic approaches. Typical number of accesses to memory range between 4 to 8. Alternately, a fast approach to perform such a lookup in hardware is by using ASICs. The ASIC based approach reduces the time by implementing *tries*. Tries are trees managed by hardware. All tree operations like adding a node, removing a node and moving tree nodes are implemented in hardware. The tree itself is stored in memory. ASIC based approaches have better performance than the software approaches but their performance is also limited by the multiple memory accesses required [13]. The typical number of memory accesses are between 4 and 8. With the advent of OC-768 (a network line capable of 40 Gbps data rate) routers have to be able to handle 40 Gbps data rate per port. By far the fastest way to implement IP routing table lookup in parallel and in hardware is by using a Ternary Content Addressable Memory (TCAM) [14, 15]. A TCAM is similar to a cache (also referred to as a CAM [16]) with the additional ability to disregard a subset of bits while performing the lookup. The prefix length of a route can be translated into a 32-bit mask, where the most significant prefix length number of bits are set to 1 and rest filled with 0. This mask is used to mask out those bits of an IP address that are not relevant to the route. TCAMs employed in routers are designed to store an entire routing table, and allow the simultaneous comparison for all routing table entries against the destination address of the packet being routed.

However, TCAMs are not without their drawbacks. They are typically expensive due to their lower production volumes. Further, they consume significantly more power than traditional RAMs. As a result, there is a strong motivation to compress IP routing tables in realistic IP routing applications. Reducing the size of routing tables would result in smaller TCAMs, thus resulting in fewer comparisons, faster IP lookup operations, lower power utilization and a lower overall system cost and maintenance

cost. Lower power consumption translates to reduced cooling requirements as well.

In this thesis, we present an algorithm which converts the routing table into a tree, and use this tree to compress the routing table. We also present an efficient algorithm to incrementally update the routing table. We exploit the fact that a route is either

- completely contained in another route,
- completely contains another route,
- is orthogonal to other routes.

Two routes never partially overlap, which means that no two routes share a portion of their address ranges. Our algorithm achieves a IP routing table compression of 24.6% on average, and is able to handle real-time route updates. It also requires less complex TCAM hardware, resulting in a 46% improvement in lookup latency and a 15% reduction in power compared to a traditional TCAM based approach.

The rest of this thesis is organized as follows. In Chapter II, we discuss preliminaries of IP routing and logic function optimization. Chapter III discusses existing literature and the causes of erroneous result in their approaches. Chapter IV introduces the new routing table compression algorithm. Chapter V details our experimental results, while conclusions are drawn in Chapter VI.

CHAPTER II

PRELIMINARIES

A. Network

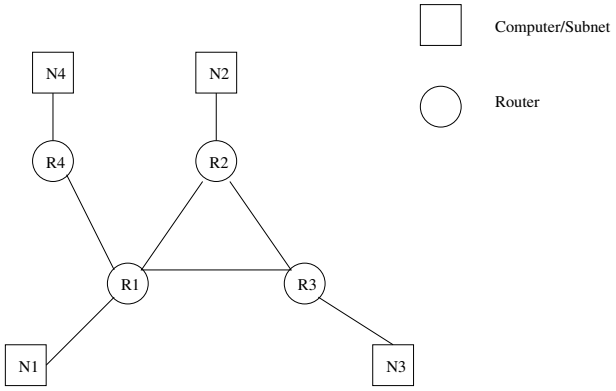

Fig. II.1. IP Routing Network Model

Consider the simplified IP routing network shown in Figure II.1. In the figure, circles indicate routers and squares indicate endpoints (either subnets or individual nodes). Suppose N1 sends a data packet intended for N2. Rather than burden N1 with a priori knowledge of the path to reach N2, the network computes a route from N1 to N2 in a localized fashion. N1 sends the data to router R1, allowing R1 to decide how to route the data to N2. R1 may now forward the data to R2, while R2 would forward the data to N2. Note that in this figure, each router has several incident edges, which are referred to as *interfaces*. So the task of each router is to forward an incoming packet on interface i to an interface j such that the packet makes "progress" in reaching its destination. A router achieves this by means of a *routing table*, which records the outgoing interface for each incoming packet. Of course, since it would be prohibitively expensive to record the outgoing interface for each destination address, routers store these entries in a compact manner. This is

achieved by storing destination addresses as *subnet prefixes* (henceforth referred to as prefixes) along with a corresponding *subnet mask* (henceforth referred to as a mask). A mask records the bits of the address that need to be considered while comparing the address with the corresponding prefix, while performing a lookup.

Table II.1 is a small IP routing table which illustrates this idea. In this table, every entry contains a subnet prefix, a subnet mask, and the *next hop* (i.e. the outgoing interface for packets whose address matches the corresponding routing table entry).

Table II.1. Routing Table Example

Prefix	Mask	Next Hop
128.96.34.0	255.255.255.128	3
128.96.34.128	255.255.255.128	4
128.96.35.0	255.255.255.0	2
128.96.34.0	255.255.255.0	5

Each time the router receives a data packet, it extracts its destination IP address and bitwise-ANDs the destination IP with the mask of each table entry (henceforth each prefix/mask entry of the table is referred to as a **route**). This operation masks out the bit positions that are not relevant to that route. The resulting data is compared with the prefix of the corresponding route. This operation is performed in parallel, in hardware. A given IP address could match multiple routes, and each of the matching routes could potentially have a different next hop. To resolve this conflict between matching routes, the "Longest Prefix Match (LPM)" rule is employed. The LPM rule sets the highest precedence to the matching route with the longest prefix length. There can only be a single route r of a given prefix length matching an

IP address, since any other route s with the same prefix length will be orthogonal to r. If a match is detected, the LPM rule is invoked to find the route with the highest precedence, and the data packet is forwarded to that route's interface (next hop).

For example, assume that a data packet with destination IP address 128.96.35.22 is received by the router from the example in Table II.1. It is ANDed with each route's mask. The resulting data from the third route matches its prefix, and therefore the data is forwarded to interface 2. If a second data packet with destination IP address 128.96.34.73 is received by the router in the example in Table II.1. The IP address would match both route 1 and route 4. This results in the LPM rule invocation. Since route 1 (prefix length = 25) has a longer prefix length as compared to route 4 (prefix length = 24), the packet is forwarded to interface 3. The LPM precedence computation is performed in hardware, either using dedicated hardware [17], or by arranging the routing table entries in a specific order [18]. The later approach could at times incur the extra cost of rearranging the entries when updates are received.

Our approach to compress a routing table utilizes Boolean logic minimization techniques. We now introduce some relevant terminology from Boolean algebra, which will be used later in the thesis.

B. Boolean Terminology

Definition 1 *An* **Incompletely Specified Boolean Function** $\mathcal{F}$ *(ISF) is a mapping*

$\mathcal{F} : B^n \rightarrow \{0, 1, *\}$

where $B = \{0, 1\}$.

Definition 2 *The* **Onset** *of an ISF* $\mathcal{F}$ *is*

$f = \{x | f(x) = 1\}$

Definition 3 *The* **Offset** *of an ISF* $\mathcal{F}$ *is*

$r(x) = 1$ *for* $\{x|f(x) = 0\}$

Definition 4 *The* **Don't Care Set** *of an ISF* $\mathcal{F}$ *is*

$d(x) = 1$ *for* $\{x|f(x) = *\}$

Note that (f, d, r) form a partition of B^n, which means that f, d, r are pairwise disjoint and together span the entire input space. In other words, for the same input x in B^n no 2 functions evaluate to 1, but at the same time at least one of them evaluates to 1 for every input x. Also, functions with $d = \phi$ are referred to as **Completely Specified Functions** (CSFs).

Definition 5 *A* **Literal** *is defined as a variable or its complement*

Definition 6 *A* **Cube** *is defined as a conjunction of literals.*

Definition 7 *A cube* c *is an* **Implicant** *of* $\mathcal{F}$ *iff*

$c \subseteq f + d$

Definition 8 *A cube* c_1 *is* **contained** *in a cube* c_2 *iff each literal of* c_2 *is present in* c_1

$c_1 \subseteq c_2 \Leftrightarrow \forall i \; l_i^{c_2} = l_i^{c_1}$

Definition 9 *A* **Sum of Products** *(SOP) expression is defined as a disjunction of cubes* $c_1 + c_2 + \cdots + c_n$.

Definition 10 *A* **cover** F *of* $\mathcal{F}$ *is a SOP expression such that*

$f \subseteq F \subseteq f + d$

There are many forms of representing a Boolean function. One of the forms that is relevant to this paper is called Sum of Products (SOP) form. In this form, the

function is represented as a list of input ranges for which the function evaluates to 1. The list is referred to as *cover* and each such range is referred to as a *cube*. A cube is represented as a combination of *literals*.

An input range for which a Boolean function evaluates to 1 can be represented as a single cube or multiple cubes, each spanning a fraction of the input. Hence, the cover representation of a function is not canonical. It has been proved that finding the exact minimal cover of a function is an NP-complete problem. Nevertheless, there exist heuristic based logic minimization techniques to minimize a cover and are known to produce near-optimal solution. The cost function for logic minimization is the number of cubes. Since we desire to minimize the number of routing table entries after compression, the cost of a routing table is the number of routing table entries. Thus, transforming a routing table into a cover, where each route is transformed into a cube, lets us apply logic minimization techniques to compress a routing table.

CHAPTER III

PREVIOUS WORK

In [19], the author proposed two techniques, called *Pruning* and *Mask Extension*, to achieve routing table compression. During the Pruning phase, the algorithm removes redundant routes. The author considers a route as redundant if there is a route with a smaller prefix length (larger range of IP addresses) existing in the routing table, which would route a packet to the same destination. The Mask Extension phase of the algorithm takes advantage of the fact that a TCAM can match an entry with any arbitrary mask. It is not required that the mask is stored in the form of a contiguous series of 1s followed by contiguous series of 0s, as is traditionally the case with routing table entries.

In [19], all routing table entries which have a given mask m and next hop n are grouped into a *cover*, in which each of the routes are translated into (Boolean) *cubes*. In IPv4 networking, prefixes and masks can be visualized as a 32-bit numbers. If each bit of the 32 bits is assigned a Boolean variable, then any combination of prefix and mask can be translated into a cube. If a mask bit $m_i = 1$ (i^{th} bit of mask m), then the literal l_i of variable x_i (i^{th} variable) is chosen to be a copy of the value p_i (i^{th} bit of prefix p) and if $m_j = 0$ then a literal l_j of variable x_j is not included in the resultant cube. The conversion of a prefix and a mask into a cube is explained with examples later in the text. This approach results in $M * N$ covers[1] and the total number of cubes over all these covers is the total number of routes in the routing table. Each of the $M * N$ covers are separately minimized using a heuristic based, two level logic minimization technique called Espresso [20].

[1]Here M is the number of distinct masks, and N the number of distinct next hops in the routing table.

Let $P(n, L)$ be the cover constructed from the routing table entries with next hop n and mask length L, and $C(n, L)$ be the cover generated by Espresso after minimizing $P(n, L)$. The result $C(n, L)$ is then transformed into a compressed routing table. For example, let us consider the routing table shown in Table III.1. Route 5 is converted into a cube $c_5 = x_1x_2x_3x_4$, while route 1 is converted into a cube $c_1 = x_1\overline{x_2}$. Route 5 belongs to $P(2, 4)$ and route 1 belongs to $P(2, 2)$.

Table III.1. Converting a Routing Table into a Cover

Entry	Prefix	Mask	Next Hop
1	1000	1100	2
2	1100	1100	2
3	1010	1110	3
4	1000	1110	4
5	1111	1111	2
6	1100	1110	4

During the pruning phase of the algorithm route 5 is considered redundant since route 2 also maps route 5's address range to the same next hop. As a result route 5 is removed from $P(2, 4)$. During the Mask Extension phase of the algorithm route 1 and route 2 are grouped into a cover (since they have the same mask length and the same next hop). Similarly routes 4 and 6 are grouped into a cover. Route 3 individually forms a cover. The resulting covers are $P(2, 2) = x_1\overline{x_2} + x_1x_2$, $P(3, 3) = x_1\overline{x_2}x_3$, $P(4, 3) = x_1\overline{x_2}\ \overline{x_3} + x_1x_2\overline{x_3}$. After these covers are formed, Espresso is run on them to compress the covers. The resultant covers are $C(2, 2) = x_1$, $C(3, 3) = x_1\overline{x_2}x_3$, $C(4, 3) = x_1\overline{x_3}$. These minimized covers are then transformed to yield the compressed

routing table of Table III.2

Table III.2. Compressed Routing Table

Entry	Prefix	Mask	Next Hop
1&2	1000	1000	2
3	1010	1110	3
4&6	1000	1010	4

With this approach, the author of [19] was able to demonstrate a routing table compression of about 45%. However, both the techniques of the algorithm can result in a corrupted routing table. The result of the resolving a packet's destination address could be different for the original routing table and the compressed (corrupted) routing table. This is a due to the *Longest Prefix Match* rule employed in routers when resolving routes containing the packet's IP address, as explained next.

Each time the router receives a data packet, it extracts its destination IP address and bitwise-ANDs the destination IP with the mask of each table entry. This masks out the bit positions that are not relevant to the entry. The resulting data is compared with the prefix of the corresponding table entry. This operation is performed in parallel, in hardware. A given IP address could match multiple routing table entries and each of the matching entries could potentially have different next hops. To resolve this conflict between matching routes, the "Longest Prefix Match (LPM)" rule is employed. The LPM rule states that the matching route with the longest prefix length wins. Since a route is either completely contained in another route, completely contains another route or is orthogonal to other routes, there can only be one route of a given prefix length matching an IP address. If a match is detected,

the LPM rule is invoked to find the winner and the data packet is forwarded to the interface (next hop) stored along with the matching table entry.

The causes of the error resulting from the use of the compressed routing table (obtained by pruning and mask extension techniques) are listed below:

- The pruning phase of the algorithm removes routes with a longer prefix, thereby losing routes with higher LPM precedence in the routing table.
- The Mask Extension phase of the algorithm could potentially combine routes to form new routes with a shorter prefix. This could result in both ambiguity and/or loss of LPM precedence in the routing table.
- Also because of the form of the routes that result from compression, the LPM rule may not be able to determine a unique winner. This results in ambiguity.

Consider the routing table of Table III.3. During the Pruning phase of the algorithm route 5 is removed due to route 2. During the Mask Extension phase routes 1&2, 4&6, 8&9, 10,11,12&13 are combined to form one route each. After the reverse transformation the resultant compressed routing table is shown in Table III.4

Packets sent to addresses 1111 and 0111 should get routed on interface 2 and interface 7 respectively because of routes 5 and 13 from the original routing table in Table III.3. Using the compressed routing table in Table III.4, they would be routed to interface 5 and interface 8 respectively because of routes 7 and 14 (since route 5 was eliminated due to Pruning, and route 13 lost its LPM precedence due to Mask Extension). A packet sent to address 1011 should get routed on interface 6 due to route 9 of the original routing table, but from the compressed routing table there is an ambiguity between route 3 and route 8&9.

The authors of [21] also follow the approach of removing routes with longer prefix by constructing a *trie* like data structure. Their approach is similar to the

Table III.3. Example Routing Table

Entry	Prefix	Mask	Next Hop
1	1000	1100	2
2	1100	1100	2
3	1010	1110	3
4	1000	1110	4
5	1111	1111	2
6	1100	1110	4
7	1110	1110	5
8	1010	1111	6
9	1011	1111	6
10	0000	1110	7
11	0010	1110	7
12	0100	1110	7
13	0110	1110	7
14	0100	1100	8

Table III.4. Example Routing Table after Compression

Entry	Prefix	Mask	Next Hop
1&2	1000	1000	2
3	1010	1110	3
4&6	1000	1010	4
7	1110	1110	5
8&9	1010	1110	6
10,11,12&13	0000	1000	7
14	0100	1100	8

pruning step of [19] but more efficient. As seen earlier, this approach could also lead to a corrupted routing table due to removal of higher precedence routes. Another approach to IP routing table compression was proposed in [22]. The authors follow a similar pruning step as [19]. After pruning the resulting routes are converted into a multi-valued logic function. This multi-valued logic function is compressed using Espresso-MV [23]. This approach compresses the routing table in a single run of Espresso-MV as opposed to multiple Espresso runs as required in [19].

Another approach to reduce the size of a routing table was proposed in [24]. This approach improves on the technique presented in [19]. In [24], authors re-introduce the routes that were pruned when routes from the minimized routing table are withdrawn. In the previous example, assume that route 7 were initially not present. Route 5 would be re-introduced into the minimized routing table if route 2 were withdrawn. This preserves the route precedence. However, route 5 is not re-introduced if route 2 were not withdrawn and route 7 were to be added. Thus ending with the same

corrupted routing table as described previously. The other improvement suggested is for reducing the time spent in running Espresso (logic minimization). Routes are grouped by next-hop and by longest common sub-prefix (LCS) rather than next-hop and mask length. The LCS for a route can be obtained by shortening the length of the prefix (reducing the number of care bits in a mask) to the nearest multiple of eight. For example, the LCS of 128.96.34.128/255.255.255.128 (entry 1 in Table II.1, with mask length of 25) is 128.96.34.0/255.255.255.0 (mask length of 24). The authors claim that the logic minimization tool has small runtime and achieves better compression when covers are formed from routes grouped by next-hop and LCS. However, this approach does not eliminate the causes of corruption in the minimized routing table as described previously. Routes with higher precedence could still be compressed to form routes with lower precedence. To summarize, the above compression approaches can cause the loss of LPM precedence and introduce ambiguity in the final routing table.

In the algorithm we present, the original routing table is always maintained in the form of a tree, and the compressed routing table is computed from the tree. Updates to the routing table are applied to the tree, and the new tree is used to compute a compressed routing table. This ensures that the resulting compressed routing table is free of ambiguity and does not lose its LPM precedence.

CHAPTER IV

OUR APPROACH

Our approach to IP routing table compression is motivated by the Mask Extension algorithm of [19]. The specific aspects that are different in our approach are

- Our algorithm operates on the original routing table which is transformed into a tree before processing.
- The resultant routing table obtained after applying our algorithm does not need a hardware unit to perform the LPM arbitration. All the routes that match are guarantied to have the same next hop. Figure IV.1 illustrates the existing architecture, where Figure IV.2 illustrates the new architecture.
- Without the need of hardware to perform LPM arbitration, lookup speeds increase, hardware cost would reduce and also power consumption would reduce as well. These gains are quantified later in this thesis.
- In our approach, finding a single match is sufficient to resolve the next hop, as opposed to finding all matches in an uncompressed routing table. Hence, in our scheme, TCAMs implemented using multiple banks could resolve multiple packets simultaneously , thereby improve average throughput.
- Our algorithm compresses all the IP address ranges with the same next hop together, irrespective of their prefix length or LCS (as opposed to [19] and [24]). This allows us to minimize a larger number of routing table entries simultaneously, thereby presenting an opportunity for better compression.

- Our algorithm to compress a routing table has a linear time complexity (as opposed to algorithm in [19] which utilizes a heuristic with a worst-case exponential time complexity).

- Our experimental results show that our algorithm can sustain the deluge of updates received during route flapping. The effect of route flapping has not been studied for any of the previously proposed approaches.

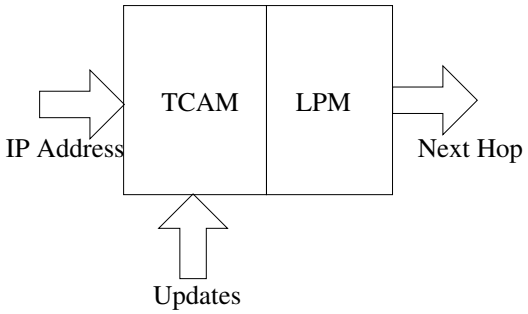

Fig. IV.1. Existing Router Architecture

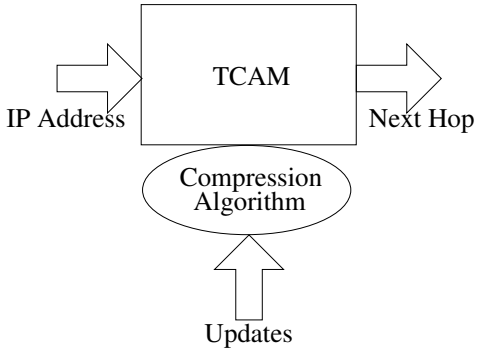

Fig. IV.2. New Router Architecture

A. Algorithm

Our algorithm exploits the fact that a route is either i) completely contained in, ii) completely contains or iii) is orthogonal to other routes. Using this fact we define the terms *parent*,*ancestor*, *child* and *sibling*. We define each of the terms in the context of a route.

A route R_p is the parent of route R if R_p is the route with the longest prefix length in the routing table that contains all IP addresses contained in R. If R_p is the parent of R then R is the *child* of R_p. A route R_a is an ancestor of route R if it contains all the IP addresses contained in R but is not the parent of R. R_s and R_t are *siblings* if they have the same parent. All siblings are orthogonal to each other. A route R_i is considered *bigger (smaller)* than route R_j if R_i contains (is contained in) R_j. A route R_i is said to be *before (after)* a route R_j if R_i and R_j are orthogonal and R_i's starting IP address (when visualized as an integer) is smaller (larger) numerically than R_j's starting IP address. The *starting address* of a route R is the numerically smallest address in the range of addresses contained in R. For example, if $p_1 = 1100$, $m_1 = 1110$, $p_2 = 1000$, $m_2 = 1100$ then $R_1 = 111-$ and $R_2 = 10--$. In this case R_2 is before R_1 since the numerical value $1000 < 1110$.

At the initialization step of the algorithm, we convert the original uncompressed routing table into a tree. Each route in the original routing table is represented by a node in the tree. Figure IV.3 illustrates an example tree structure for a routing table with 8 entries. The tree has one node per routing table entry. There is a directed edge from node C_i to node C_j if C_i is a parent of C_j.

However to improve efficiency , the tree data structure is implemented differently. Figure IV.4 illustrates the implementation of the tree. Each child has a pointer to the parent (not shown in the figure). The default route is at the root of the tree and

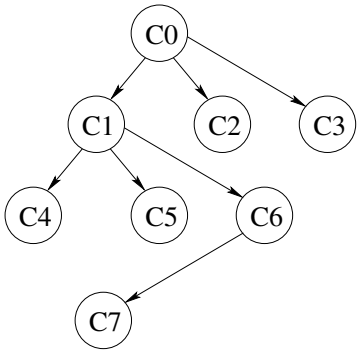

Fig. IV.3. Tree Form of a Routing Table

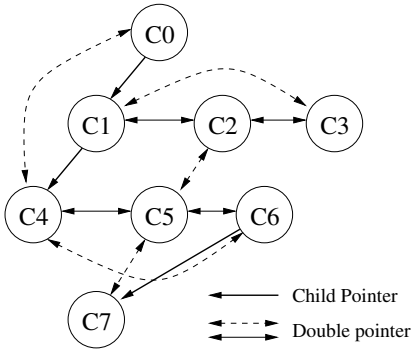

Fig. IV.4. Tree Data Structure Realization

has no parent or a parent pointer. All children of a route are part of a doubly linked list (shown as the solid double arrow). The parent has a pointer to the double linked list of its children (shown as the solid child pointer with a single arrow). This doubly linked list is sorted in the decreasing order of route's starting IP address.

The use of the tree structure ensures that the precedence of routes is preserved (while computing the compressed routing table) in the form of its position in the tree. A node with no children (leaf node) has the highest precedence on its address space, followed by it's parent then by it's parent's parent and so on. A node with no parents is referred to as the root node and such a node is the default route of the routing table.

Each node is also part of a second doubly linked list maintained per next hop (shown as the broken line double arrow). Each node in the tree additionally has a hash table of 32 buckets (not shown in the figure). The IP address range spanned by a route is equally distributed into these buckets. All child nodes that fall into a bucket's subrange are inserted into that bucket. Using the hash table reduces the average time complexity to search for a route in the tree. Note that a child's range could be larger than that of a bucket in which case the child is put in all the buckets that it spans.

We define the following terms which will be used in the remainder of this section:

- $D[i] \rightarrow$ cover of the routes with the same next hop 'i'.
- nodeList[i] $\rightarrow$ doubly linked list of nodes in the tree with next-hop 'i'.
- *Node cover* $E[n]$ of a node n is the cover of IP addresses that belong to node n and not to it's children. This is computed as $E =$ cube of node n - cubes of children, where '-' is the set difference operator.

- The *distance* between two cubes is defined as the number of literals that the cubes differ in.

- *Size* of a cube is computed from the number of literals in the cube. As the number of literals increases the size of the cube decreases.

We now present the algorithm to search for a route in the tree. The algorithm is recursive. It starts at the root node and recursively calls itself until the required node is found. Algorithm NODESEARCH (Fig. IV.5) illustrates the approach. The algorithm takes two inputs, node n whose sub-tree is to be searched and route r whose corresponding node n_r is to be located in the tree. On line 1 we extract the set of children *childSet* from the hash bucket that could potentially contain node n_r. This set is identified by the range of IP addressed spanned by route r. Line 2 loops over all the children in the childSet. Each child c's route c_r is compared with route r. The comparison has five possible outcomes:

- Route c_r is bigger than r. In which case, node c is an ancestor of node n_r. Thus, on lines 3 and 4 we perform the check and search the sub-tree rooted at node c for node n_r.

- Route c_r is smaller than r. This indicates that node n_r does not exist in the tree. If it were to exist, it would have been an ancestor/parent of node c. If there were a smaller node than node n that contains n_r, then it would be the parent of node c. Thus we identify node n as the parent of node n_r. Lines 5 and 6 perform the check and return node n.

- Route c_r is same as route r. In which case, node c is returned in lines 7 and 8.

- Route c_r is before route r. As mentioned earlier, the list of children are sorted in the decreasing order of their starting IP address. If route c_r is before r then

```
  Algorithm nodeSearch (n, r)
Input:
    n - node whose sub-tree is searched
    r - the route to be searched
 1      childSet ← hashBucket[hash(r)]
 2      foreach child of childSet do
 3        if child bigger than r then
 4          return nodeSearch(child, r)
 5        else if child smaller than rthen
 6          return n
 7        else if child = rthen
 8          return child
 9        else if child before rthen
            /* since child list is sorted in descending order */
10          return child
         /* else child is after the route r */
         /* continue with next child */
```

Fig. IV.5. Pseudo Code to Search the Tree

node n_r does not exist in the tree. If it were to exist in the tree then, it would have been encountered before reaching node c. Thus, node n is identified as node n_r's parent and node n_r is to be placed before node c in the linked list of children in lines 9 and 10.

- Route c_r is after route r. In which case, we continue searching through the child list.

The worst case time complexity of the search algorithm is of the order of the number of nodes in the tree. But owing to the nature of the problem, certain properties of the tree can be used to improve the average time complexity of the algorithm. The prefix length of a child node has to be more than its parent's prefix length. From this fact we can conclude that the height of the tree is bounded (by 32 in case of

IPv4). Hence, the tree is much wider than it is deeper. Due to this fact, the average time complexity of the search algorithm is heavily dependent on the size of the hash table in each node. A hash table with 32 buckets was picked empirically.

B. Longest Prefix Match

In a typical router, routing table updates are less frequent than the routing table lookups. Since LPM arbitrations are performed in hardware, this incurs extra hardware cost and more power consumption. If routing table entries were preprocessed to ensure that any two routes with different next hops were orthogonal to each other, i.e., to ensure that all routes matching an IP address have the same next hop, the router can be implemented without an LPM hardware unit. To convert the original routing table into the above mentioned form, we subtract the IP address ranges spanned by a route's children from the route's IP address range. The result is then converted to cubes and the cover formed from the cubes is stored in the node. Additionally the cube representation of the original route is also stored in the node.

In Boolean algebra the result of subtracting a cube c_2 from another cube c_1 is computed as $c_1 \setminus c_2 = c_1 \cdot \overline{c_2}$. For example let $c_1 = 1$ - - - and $c_2 = 1001$. The value of $\overline{c_2}$ is all possible values except c_2, thus $\overline{c_2} = 0$ - - - + - 1 - - + - - 1 - + - - - 0. The result of subtracting c_2 from c_1 is 11 - - + 1 - 1 - + 1 - - 0, which represents all possible values in c_1 but not in c_2. A second subtraction could potentially multiply the size of the cover by the number of literals in the cube. Subtracting multiple cubes from a single cube could potentially increase the run time complexity and the size of the cover exponentially. To be able to subtract child routes from a route efficiently, we propose a linear time algorithm. A route's IP address range can be visualized as a series of integers on a number line. Subtracting IP address ranges of its children can

be performed in linear time, but the resultant ranges after subtraction can not always be represented as cubes. For example, consider a route and its child in Figure IV.6.

$$\begin{array}{rc} & 1000{\leftrightarrow}1111 \\ - & 1001 \\ \hline & 1000 + 1010{\leftrightarrow}1111 \end{array}$$

Fig. IV.6. Example Route Subtraction

Subtraction results in a set of IP address ranges, some of which (1000) can be expressed as a single cube and some (1010↔1111) that can not be expressed as single cube. To keep the total run time bounded, we developed an algorithm to convert these ranges to cubes in constant time (Algorithm coverRange).

The algorithm takes the lower bound (*low*)and upper bound (*high*) of the range of IP addresses as inputs. The inputs are converted to 32-bit numbers (IPv4). The operation on line 3 results in a value in which all the contiguous least significant zeros of *low* are 1. The range of IP addresses from *low* to *lhigh* can be represented as a single cube z. From the example in Figure IV.6, $low = 1010$ and (low-1) = 1001, so $low|(low\text{-}1) = lhigh = 1011$ and the cube representation of z is 101-. Similarly, if $low = 1100$ then (low-1) = 1011, so $low|(low\text{-}1) = lhigh = 1111$ and the cube representation is 11–. The quantity $lhigh + 1$ is guarantied to have more least significant zeros than *low*, so across iterations the value of *low* increases and has more least significant zeros. The boundary case when $lhigh = 1111$ needs special handling as shown in lines 6 and 7. When *lhigh* is larger than *high*, smaller cubes are constructed to cover the remaining range of addresses. This ensures that the iteration count is bounded (by 64 in case of IPv4). Since each step in the loop takes constant time, the total time complexity is constant. Also, since an IP address is covered only by one

```
  Algorithm coverRange (low,high)
Input:
   low - start of the IP address range
   high - end of the IP address range
1       F ← φ
2       while low ≤ high do
3         lhigh ← low|(low-1)
4         if lhigh ≤ high then
5           F ← F ∪ formCube(low,lhigh)
6           if lhigh == INT_MAX /* avoid overflow */
7             return F;
8           low ← lhigh + 1
9         else
10          range ← (lhigh⊕low)≫1
11          while low+range > high do
12            range ← range ≫ 1
13          F ← F ∪ formCube(low,low+range)
14          low ← low + range + 1
15      return F
```

Fig. IV.7. Pseudo Code to Convert a Range of IP Addresses to Cubes

cube in the result, all the cubes in the result are orthogonal to each other.

By using the Algorithm COVERRANGE, each range of IP addresses can be represented by a bounded number of cubes and the number of IP address ranges is of the order of the number of children. So the number of cubes in cover E is of the order of the number of children. Also, the time taken to compute cover E constitutes the time taken to subtract a node's children from it and the time taken to convert the resulting IP ranges into a cover. Both operations have a time complexity of the order of the number of children. So the total time complexity is also linear in the number of children.

C. Efficient Logic Minimization

We now discuss our linear time algorithm to reduce the size of a routing table. Finding the exact solution for 2-level logic minimization has been proved to be NP-Complete. Heuristic based logic minimization algorithms result in near optimal solutions but have exponential runtime in the worst case. The existing algorithms are designed to work with covers that contain overlapping cubes and can not therefore take advantage of the nature of problem at hand. The cover of a next hop is orthogonal to the cover of every other next hop. Also, every cube in the cover of a next hop is orthogonal to other cubes in the cover. Hence, every cube is orthogonal to every other cube. For such covers we observed that pairwise merging of cubes results in significant compression with linear time complexity. The linear time complexity of our algorithm allows us to handle real-time updates to the routing table, including route flapping scenarios.

Algorithm D1COMPRESS (Fig. IV.8) presents our approach. The *d1merge* routine used in the algorithm is a simple cube merging routine that merges two cubes of the same size, differing in exactly one literal. Such cubes have a Hamming distance of

```
  Algorithm d1compress (F)
Input:
   F - A cover of pairwise orthogonal cubes
 1        G ← φ
 2        for i in 32 → 1 do /*for IPv4 */
 3          G ← G ∪ cubesOfSize(F, i) /*O(n)*/
 4          G ← sort(G) /*O(n) Radix Sort*/
 5          c1 ← first cube in G
 6          foreach cube c2 in G do /*O(n)*/
 7            if can d1merge c1 and c2 then
 8              *c1 ← d1merge(c1,c2) /*replace contents of c1 */
 9            else
10              c1 ← c2
```

Fig. IV.8. Pseudo Code to Compress a Cover Using *d1merge*

1. If two cubes have a Hamming distance of 1, then the cube formed by removing the conflicting literal is the result of merging the cubes. For example, the two cubes 10-1 and 00-1 differ in exactly one literal. The result of merging the two cubes is -0-1. The D1COMPRESS algorithm starts with cubes with the highest literal count (32 literals) and pairwise merges as many of them as possible (forming cubes with 31 literals), after which cubes with one literal less than the highest literal count (31 literals) are merged. Processing cubes in decreasing order of literal count helps newly merged cubes to find more potential cubes of same size to merge with. Also, to make sure that cubes that can be merged are ordered next to each other, they are sorted based on their literal count and then lexicographically. The basis for comparing two cubes in the sort procedure on line 4 is "-" < "0" < "1".

```
  Algorithm compress ()
1       foreach node n in tree do
2         compute cover E of node n
3       foreach nextHop do
4         F ← φ
5         foreach node in nodeList[nextHop] do
6           F ← F ∪ cover E of node
7         D[nextHop] ← d1compress(F)
```

Fig. IV.9. Pseudo Code to Compress the Routing Table

D. Compression

We now explain the algorithm to compress a routing table. Algorithm COMPRESS illustrates our approach. We generate node covers for every node in the tree in lines 1 and 2. The cover for node n is obtained by subtracting the cubes of all children of n, from the cube of node n. In every iteration of line 3 we accumulate node covers of nodes that map to the same next hop (lines 4, 5 and 6). In line 7 we compress the accumulated cover and store it in the TCAM.

The time complexity of line 2 is of the order of the number of children of node n (as discussed in Section IV.B). The time complexity of lines 1 and 2 together is of the order of the total number of children in the tree. Since a node is the child of only one node in the tree, the total number of children in the tree is equal to the number of nodes in the tree. Thus, the time complexity of lines 1 and 2 is of the order of the number of nodes in the tree. The time complexity of each iteration of line 6 is of the order of number of children in the node (since E is of the order of children). The total time complexity of lines 5 and 6 in all iterations of the loop on line 3 is of the order of all the children in the tree. The total number of children in the tree is same as the total number of nodes. So the time complexity of lines 5 and 6 is also of the order of

the number of nodes in the tree. By a similar argument, the total time complexity of line 7 in all the iterations of line 3 is also of the order of the number of nodes in the tree. Note that the time complexity of Algorithm D1COMPRESS is linear in the size of input. From the above discussion we can conclude that the total time complexity of Algorithm COMPRESS is of the order of the number of nodes in the tree.

E. Updates

Routers typically receive millions of updates per day. Routers going off-line withdraw the routes handled by them, and routers coming on-line update other routers about the routes available through them. Any router should be able to keep up with the volume of updates it receives from other routers. Routers using the algorithm being proposed have to be able to apply these updates in real time. An update has to guarantee that all the covers in the routing table are still orthogonal to each other and their routing precedence is maintained. To ensure orthogonality and to be able to find the changes in precedence due to the update, the tree structure has to be updated along with other data structures. To be able to apply the updates in real time we propose two highly efficient algorithms to insert and withdraw routes.

1. Insertions

Algorithm INSERT illustrates our approach to insert new routes into a routing table. Line 1 searches for the location of the new route in the routing table. Lines 2-7 insert the new route in the tree, if the new route is not a duplicate of an existing route. Inserting or withdrawing a route could change the precedence of existing routes. Once the route is inserted, node covers have to be computed again. Inserting a new route does not change the precedence of any route with a longer prefix than the new route.

Hence, the node cover of any node with a longer prefix need not be updated. All the routes that are not ancestors of the inserted route do not share the IP address space with the inserted route. Hence the node covers of any of the routes that are not ancestors of the inserted route need not be updated. All the ancestors of the inserted route must have lost precedence over the inserted route's space to the route's parent. Thus, the only node covers that need to be computed are those of the inserted route and its parent. Lines 8 and 9 compute the covers of the inserted node and its parent respectively. After updating the node covers, the TCAM has to be updated with the changes in the routing table. If the inserted route and its parent have the same next hop, then the TCAM need not be updated. However, if their next hops differ then the only next hop covers in TCAM that need to be modified are those of the new route's next hop and its parent's next hop. Lines 10-13 compute the next hop covers and update the TCAM. The $\setminus$ operator is the set difference (or subtraction) operator over covers.

Line 8 presents a way to compute the cover of the inserted node. Alternatively, $E[m]$ can be computed from the new node's children (lines 1 and 2 of Algorithm COMPRESS). In our implementation of the INSERT algorithm, we heuristically choose between these two methods to compute $E[m]$. The first method is chosen whenever the number of children of the newly inserted node is greater than half the number of cubes in parent's cover, otherwise we select the second method. The worst case time complexity of the INSERT algorithm is bounded by the order of the number of nodes in the tree. Though the time complexity is linear in the order of the size of the tree, such a complexity is unacceptable. The average run time of the algorithm is much better than the worst case time complexity.

```
  Algorithm insert (root, r)
Input:
    root - The root node in tree representation of the routing table.
    r - The new route to be added.
1         n ← nodeSearch(root, r)
2         if n == r then
3           return /* duplicate update */
4         else if n bigger than r then
           /* insert in child list of n */
5           m ← insertUnder(n, r) /* m is the inserted node */
6         else if n is before r then
           /* insert before n, since child list is sorted in
           descending order */
7           m ← insertBefore(n, r) /* m is the inserted node */

         /* update covers in the nodes */
8         E[m] ← E[parent(m)] ∩ cube(m)
9         E[parent(m)] ← d1compress(E[parent(m)] \ cube(m))
         /* update next hop covers */
10        i ← nextHop(m); j ← nextHop(parent(m))
11        if i ! = j then
12          D[j] ← d1compress(D[j] \ cube(m))
13          D[i] ← d1compress(D[i] ∪ E[m])
```

Fig. IV.10. Pseudo Code to Insert a Route into the Routing Table

2. Withdrawal

Algorithm WITHDRAW illustrates our approach to withdraw a route from the routing table. Line 1 searches for the route in the routing table. Lines 2-7 remove route from the tree, if such a route exists in the tree. After withdrawing a route n, n's parent will be the smallest route containing all the children of node n. On line 6 n's parent becomes the new parent of its children. Once the route is removed, node covers have to be updated. Removing a route does not change the precedence of any route with a longer prefix. Hence, the node cover of any route with a longer prefix need not be updated. All the routes that are not ancestors of the withdrawn route do not share the IP address space with the withdrawn route. Hence the node covers of any of the nodes that are not ancestors of the withdrawn node need to be updated. Also, all the ancestors of the removed node will have less precedence over the removed route's space compared to the route's parent. So the only node cover to be re-computed after the removal is the node cover of the parent. Line 8 computes the cover of the parent. After updating the node cover, the TCAM has to be updated with the changes in the routing table. If the removed node and its parent have the same next hop, then the TCAM need not be updated. However, if their next hops differ then the only next hop covers in TCAM that need to be modified are those of the removed node's next hop and its parent's next hop. Lines 9-14 compute the next hop covers and update the TCAM.

The worst case time complexity of the WITHDRAW algorithm is bounded by the order of the number of nodes in the tree. Though the time complexity is linear in the order of the size of the tree, such a complexity is unacceptable. The average observed run time of the algorithm is much better than the worst case time complexity.

```
  Algorithm withdraw (root, r)
Input:
    root - The root node in tree representation of the routing table.
    r - The route to be withdrawn.
 1        n ← nodeSearch(root, r)
 2        if n ! = r then
 3          return /* no such route exists*/
 4        else
 5          foreach child of n do
 6            parent(child) ← parent(n)
 7            add child to parent(n)'s child list
         /* update covers in the nodes */
 8        E[parent(n)] ← d1compress(E[parent(n)] ∪ E[n])
         /* update next hop covers */
 9        i ← nextHop(n); j ← nextHop(parent(n))
10        if i ! = j then
11          D[j] ← d1compress(D[j] ∪ E[n])
12          F ← φ
13          foreach node in nodeList[i] do
14            F ← F ∪ E[node]
15          D[i] ← d1compress(F)
```

Fig. IV.11. Pseudo Code to Withdraw a Route from the Routing Table

F. Optimizations

In algorithm D1COMPRESS on line 4, radix sort was used to achieve a time complexity linear in the size of input. We observed that due to a majority of relatively small cover sizes, replacing radix sort with quick sort resulted in better run times.

Each update involves a search in the tree followed my updating the compressed port covers ($D[\,]$). Our experiments indicate that the time per update is dominated by the computation of compressed port covers. In an attempt to reduce the overall runtime of the algorithm and to be able to handle updates in real-time, we perform a second optimization. When an update does not increase the size of the port cover by more than 2% we do not invoke the compression algorithm. This approach results in better real-time performance of the system but results in a gradual reduction in the compression achieved, as time progresses.

CHAPTER V

EXPERIMENTAL RESULTS

The algorithms discussed in the previous section were implemented in the C programming language. The open source GNU C compiler was used to build the binaries. Routing Information Base (RIB) files and the corresponding update files were downloaded from [25]. All experiments were run on a single core of Intel(R) Core(TM)2 Quad Core CPU operating at 2.66 GHz with a system memory of 4GB. The results of our experiments are presented in Table V.1.

The first column contains the source and date of the routing table being compressed. The routers located at

- University of Oregon, Eugene, Oregon, USA (U of Ore.)
- Equinix, Ashburn, VA (eqix)
- ISC (PAIX), Palo Alto CA, USA (paix)

were chosen as data sources. The second column lists the original number of routing table entries at the beginning of the day in question. The third column reports the number of entries in the routing table after compression during the initialization phase of the algorithm (and the % compression achieved). The fourth column reports the time taken in seconds to compress the routing table during the initialization phase. Column 5 reports the total number of updates received by the router on that day in a 24 hour period (and the average number of updates per second). Column 6 reports the average time taken to propagate the effect of the update into the TCAM in seconds. Column 7 reports the size of an uncompressed routing table at the end of the 24 hour period. Column eight reports the size of the compressed routing table at the end of the day (and the % compression at the end of the day). Column nine

Table V.1. Experimental Results

Router/ Date	Original Size	Compr. Size (%)	Time to Compress (s)	# of Updates (/sec)	Time /update (s)	Final Size	Final Compr. Size (%)	Total Run Time (s)
U of Ore. 11/21/2008	289561	215929 (25.43)	3.332494	10401712 (120.39)	0.000746	289626	232343 (19.78)	1611.53
U of Ore. 11/20/2008	288523	214442 (25.68)	3.311497	10399463 (120.36)	0.000322	289561	227284 (21.51)	2062.71
U of Ore. 11/19/2008	286069	211919 (25.92)	2.688591	12165311 (140.80)	0.000532	288523	251831 (12.72)	1535.75
U of Ore. 11/18/2008	285539	211790 (25.83)	2.688591	11335607 (131.20)	0.000338	286069	226145 (20.95)	1139.15
U of Ore. 11/17/2008	286977	212206 (26.06)	2.699589	9452658 (109.41)	0.001446	285539	223134 (21.86)	2051.30
eqix 11/21/2008	274545	204683 (25.45)	2.504619	10694627 (123.78)	0.000841	274507	264281 (3.73)	8048.42
eqix 11/20/2008	273969	203635 (25.67)	2.496621	13979295 (161.80)	0.000819	274545	266250 (3.02)	10073.93
eqix 11/19/2008	270995	202942 (25.11)	2.423632	15634233 (180.95)	0.000807	273969	264946 (3.30)	11039.58
eqix 11/18/2008	270684	202718 (25.11)	2.432631	10152552 (117.51)	0.000822	270995	264655 (2.34)	7251.50
eqix 11/17/2008	270302	202628 (25.04)	2.411633	13333514 (154.32)	0.000837	270684	264290 (2.36)	10149.40
paix 11/21/2008	276922	214215 (22.64)	2.609603	1866871 (21.60)	0.000457	279252	223429 (19.99)	205.24
paix 11/20/2008	276394	213647 (22.71)	2.603605	2328720 (26.95)	0.000332	276992	220158 (20.50)	217.79
paix 11/19/2008	276426	213799 (22.66)	2.592606	2836798 (32.83)	0.000264	276394	221389 (19.90)	221.82
paix 11/18/2008	275710	213771 (22.47)	2.592606	2396366 (27.73)	0.001985	276426	219936 (20.44)	552.10
paix 11/17/2008	275564	213636 (22.48)	2.589606	2004843 (23.20)	0.001301	275710	218598 (20.72)	426.67
Average	278545.33	210130.66 (24.55)	2.665195	8598838.00 (99.52)	0.000790	279252.80	239244.60 (14.21)	3772.46

reports the cumulative CPU time consumed by the approach in seconds, over the 24 hour period.

Figure V.1 and Figure V.2 plot the size of the uncompressed routing table and

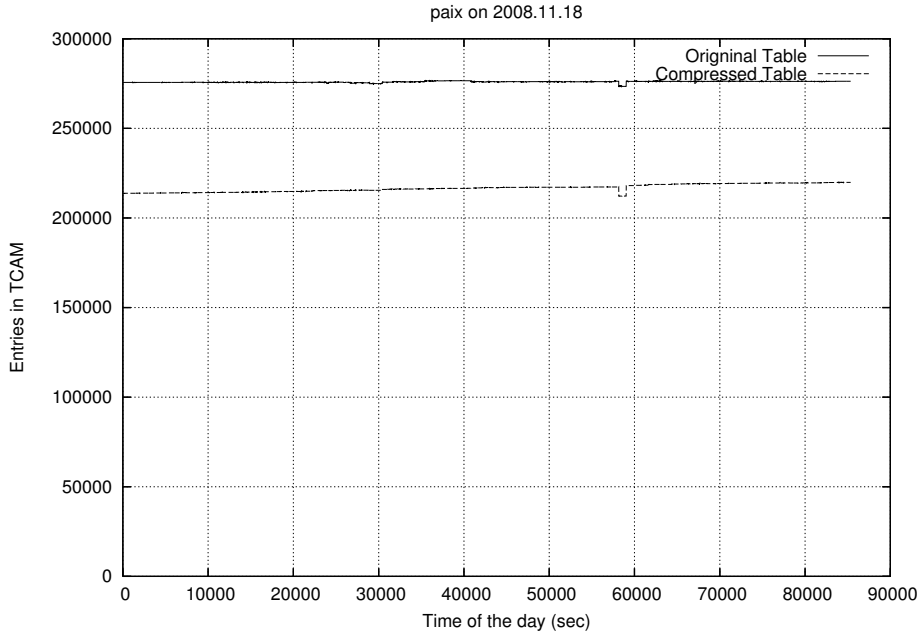

Fig. V.1. Table Sizes Example 1

the size of the compressed routing table during a 24 hour period on 11/18/2008, for the PAIX router, and on 11/21/2008 for the University of Oregon routers respectively. These plots are representative of the results obtained for similar experiments on different routers and different days. We observe that on average, our algorithm can compress the routing tables to about 24.55% of its original size. The maximum time for the initial compression is less then 3 seconds. The maximum time to update the compressed routing table is about 0.2 seconds, while the average time per update was also observed to be less than 0.0008 seconds. Figures V.1 and V.2 show a gradual increase in the size of the compressed routing table. This is because the compression heuristic is not invoked when the change in the size of the cover is less than 2%. This gradual increase in table size can be corrected by using the algorithm COMPRESS

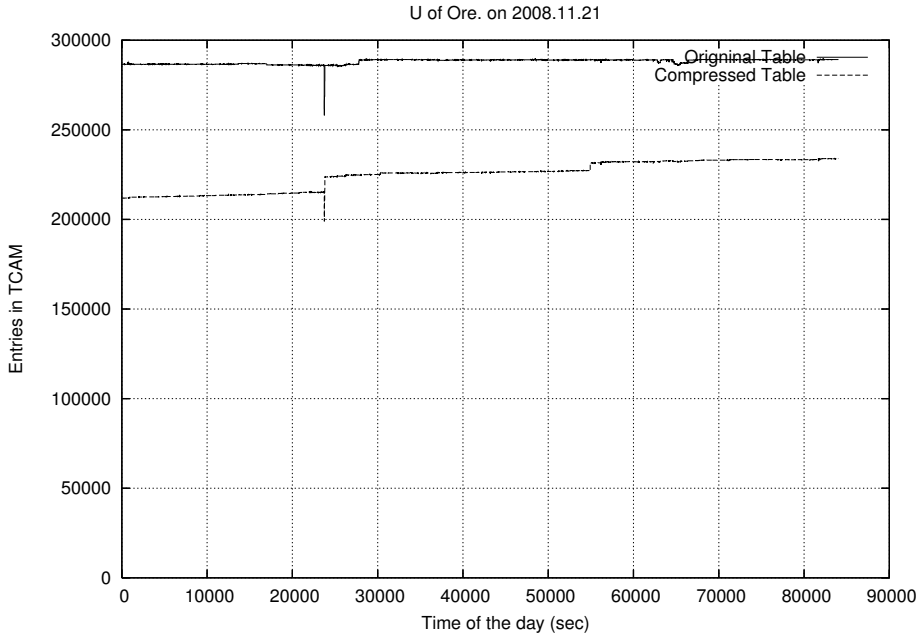

Fig. V.2. Table Sizes Example 2

during times of the day when the load on the router is at a minimum.

Figures V.3 and Figure V.4 are scatter plots that plot the time taken to process an update verses the time until the next update arrives. The examples used in these plots are the same as those used in Figures V.1 and V.2 The diagonal dotted line in these plots indicates all the points whose processing time was equal to the time until the next update. All the points below this line represent updates that were processed before the arrival of the next update. All the points above the line represent updates that needed more processing time than the time of the arrival of the next update. We observe that the majority points are below the diagonal line. Consequently the total run time in Column 9 of Table V.1 is far less than 24 hours.

A. Compared to Espresso

We compare the performance of our routing table compression heuristic with Espresso, (which has exponential time complexity in the worst case). On an average, Espresso

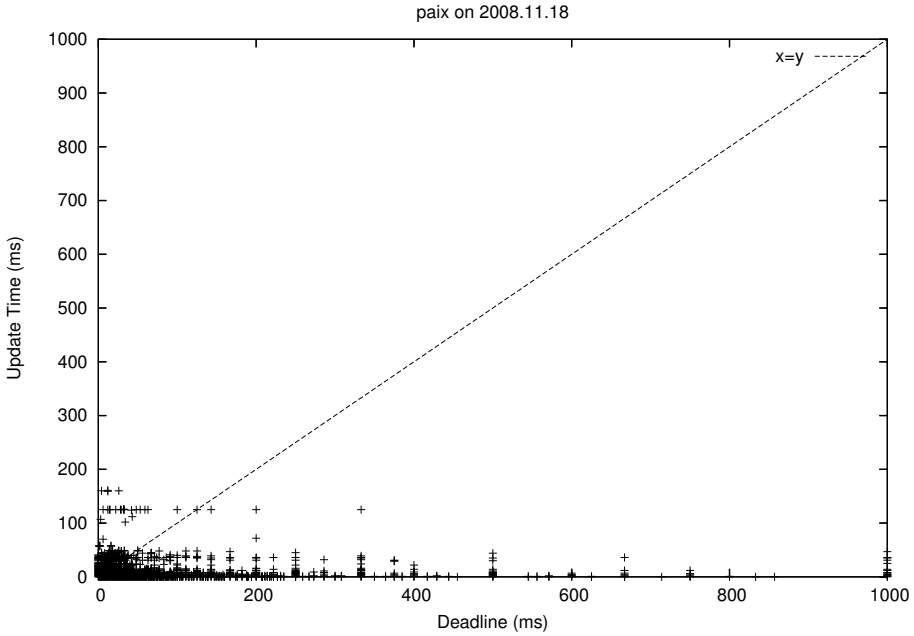

Fig. V.3. Scatter Plots of Example 1

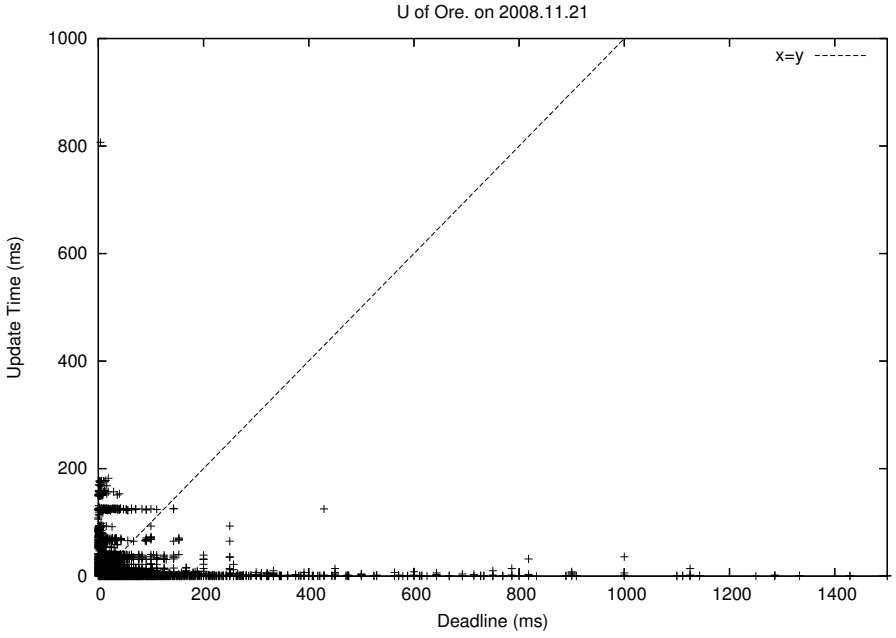

Fig. V.4. Scatter Plots of Example 2

achieves 38% compression in the size of the routing table in stead of 24.55% compression from our linear time heuristic. However, Espresso requires a 20× increase in runtime during the initial phase of the algorithm (when the original tree is compressed). Also the total cumulative time taken to handle updates is 16.8× higher for Espresso than our approach. This increase in run-time is with all the optimization described in the Section IV.F. Figure V.5 illustrates the size of the uncompressed and the compressed routing tables when Espresso is used to compress covers, for the University of Oregon router on 11/19/2009. Note that a higher compression is achieved with Espresso.

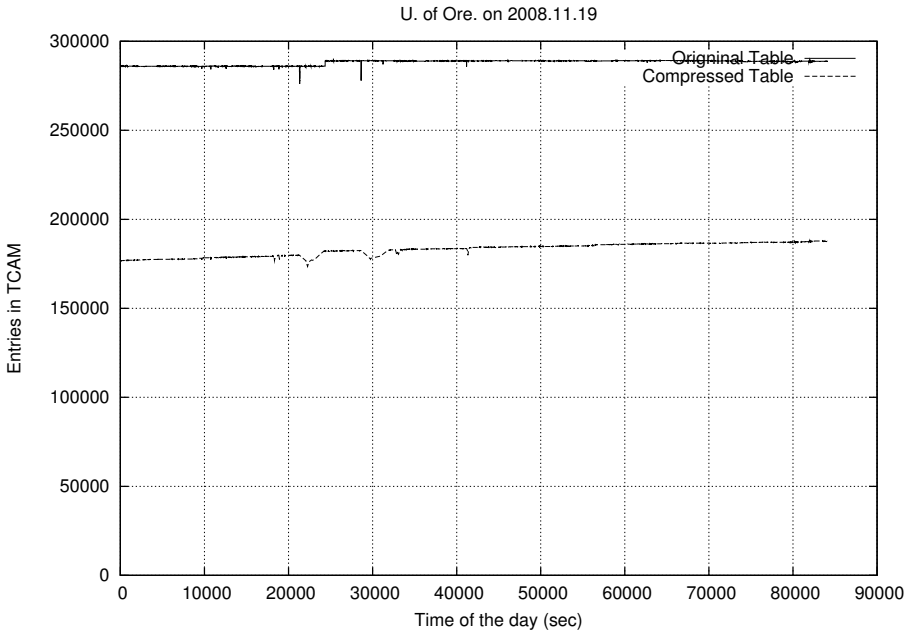

Fig. V.5. Routing Table Size Employing Espresso

Although Espresso provides better compression than our compression heuristic, it has unacceptable real-time performance. Figure V.6 plots the delay in processing an update. The x-axis of the graph is the time during the day. Let us consider an update received a time t. The value δt at time t in the graph is the time after t when the affect of the update was written into the TCAM. In other words, the update

received at time t was applied at time $t + \delta t$. This delay is due to the occurrence of updates in quick succession, hence requiring more run-time than the difference in their arrival times. The accumulated delay in processing all the updates before the i^{th} packet causes the delay in processing the i^{th} packet. As can be observed, some of the updates are delayed by more than 50 minutes, which is unacceptable. The maximum delay incurred in our heuristic for this example was observed to be less than 6 seconds.

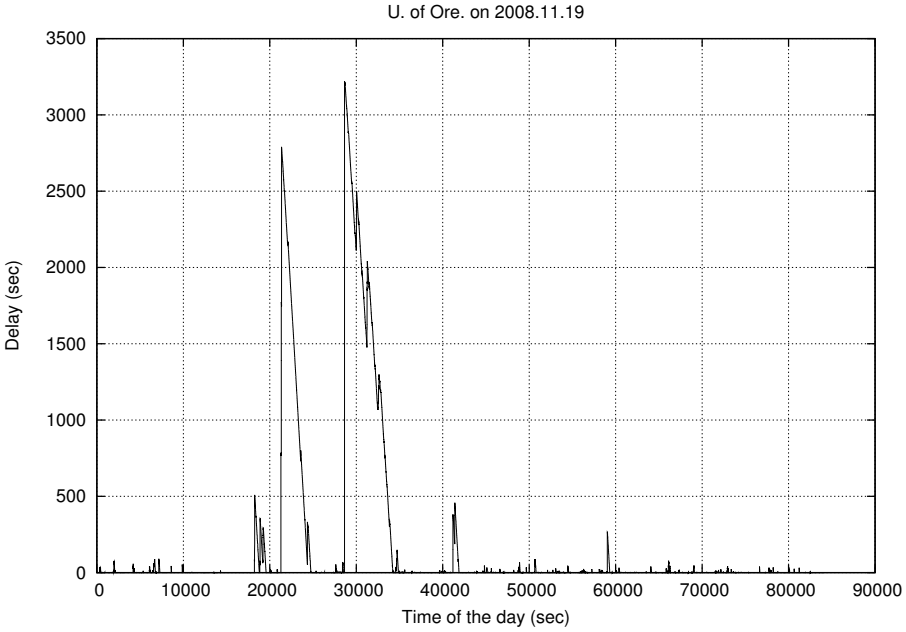

Fig. V.6. Espresso's Processing Delay

Figure V.7 plots a distribution of the time to process an update (on y-axis) verses the time until the next update arrives (on x-axis). This plot and Figure V.4 are generated from the same example. The diagonal dotted line in the plot indicates all the points whose processing time was equal to the time until next update. Points below the line are updates that were handled before the next update and points above the line are updates that were not handled before the next update. We observe that in the Espresso based approach a majority of the updates could be successfully processed

after the next update arrives. The height of a point above the dotted line indicates the additional time required. This additional time accumulates accumulates across successive updates and results in the processing delays similar as shown in Figure V.6.

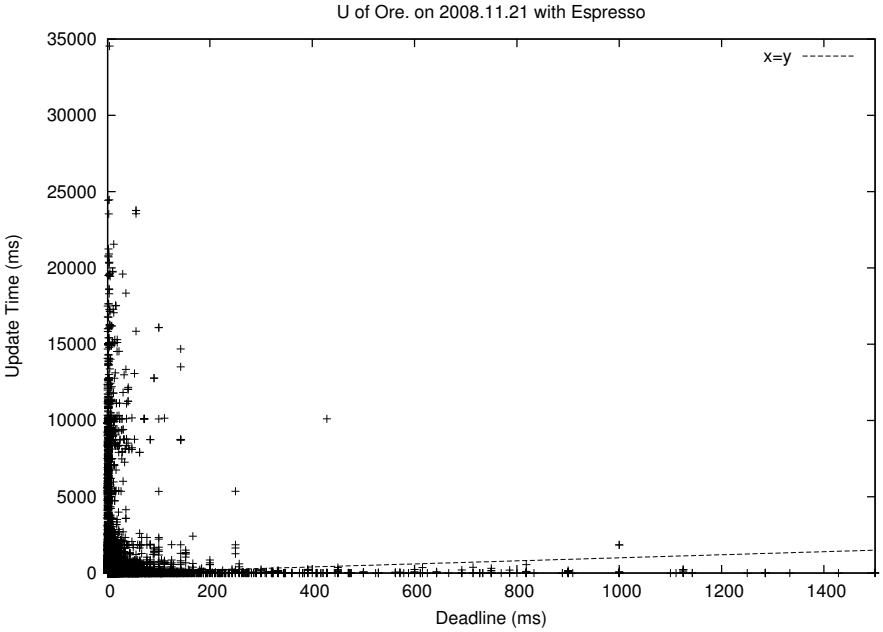

Fig. V.7. Scatter Plot of Espresso Approach

The real-time performance of our heuristic can be better judged by its performance under route flapping. We now describe and analyze the performance of our heuristic during route flapping.

B. Route Flapping

All routing tables, are severely effected by route flapping and interface flapping which occur due to updates from neighboring routers. When a router is not configured correctly on the network, or if there is a malfunction in the hardware of the router, routes can be withdrawn and re-announced in quick succession. This causes the size of the routing table of its neighbors to fluctuate rapidly. The number of updates received during such a period could exceed 1250 updates per second, which can cause

a significant delay in processing updates. A router should be able to handle such a deluge of updates, and still be able to correctly route packets towards their destinations. Routers implementing a compression algorithm are adversely affected by route flapping, since every update received during route flapping has to be compressed, and the result has to be updated into TCAMs. On 18^{th} November 2008 the router *eqix* experienced such a phenomenon. Our experiments show that a router implementing our algorithm is capable of sustaining the load due to route flapping. Figure V.8 shows the fluctuations in the routing table size. Note that the uncompressed routing table size (sold line) varies between 271000 routes and 22000 routes several times during the day, and our compressed routing table (dotted line) keeps up with these fluctuations.

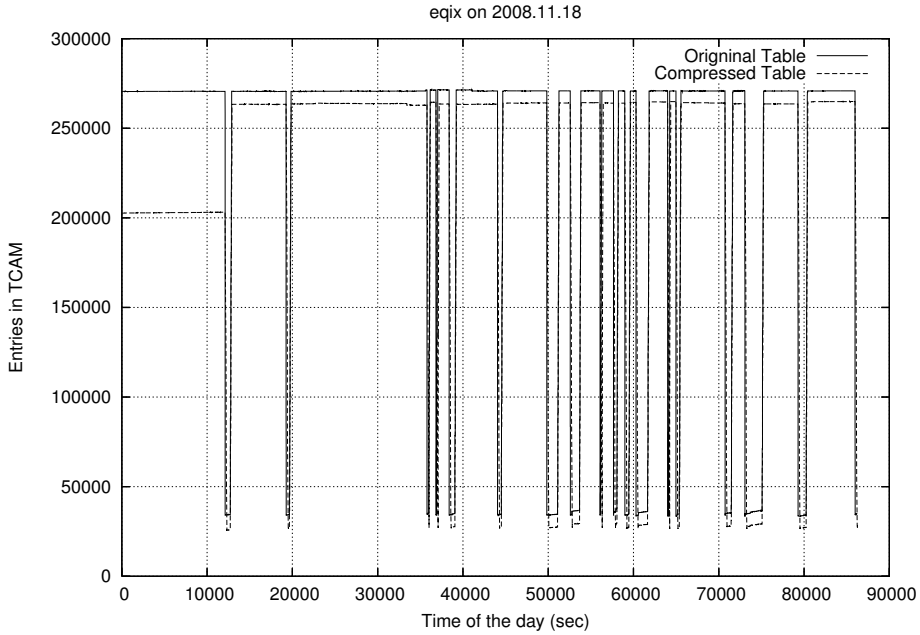

Fig. V.8. Flapping Example

Figure V.9 shows the delay between the arrival of an update and the time when its effect is updated in the TCAM We observe that our algorithm can keep up with the heavy fluctuations in table size with a maximum delay of 250 seconds in processing

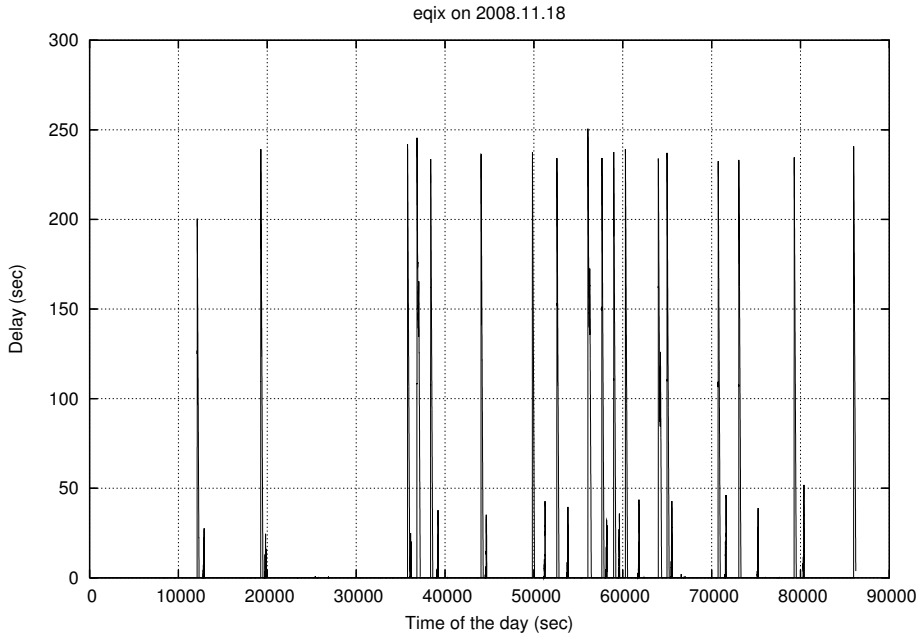

Fig. V.9. Flapping Example's Processing Delay

an update.

C. Hardware Cost Reduction

As mentioned earlier in Section IV.B LPM computation is no longer required because compressed routes for any two next hops are guarantied to be orthogonal. This can be utilized in simplifying the TCAM used. Compared to [26], this simplified TCAM yeilds a reduction in lookup time by 49%, power by 9% and area by 9%. These saving are from simplification of the hardware alone. There would be further saving in the form of reduced size of TCAM required in storing the compressed routing table. TCAMs like other memory arrays are implemented in banks. A 15% reduction in routing table size (average compression at the end of the day in Table V.1) translates to fewer TCAM banks or smaller TCAM banks. In both the cases power and area can be saved.

D. Scaling

As shown earlier, our approach can handle the size of current routing tables, but it is important to analyze the performance as the global IP routing table continues to grow super-linearly. Figure V.10 plots the size of the uncompressed routing tables (solid line) and the compressed routing tables (dotted line) from year 2001 to till date, for the University of Oregon router. As the size of the global routing table increases the percentage compression achieved increases, thus demonstrating that our approach is able to contain the super-linear growth of routing tables. Using our approach, hardware resources need not scale at the rate of growth of the uncompressed global IP routing table.

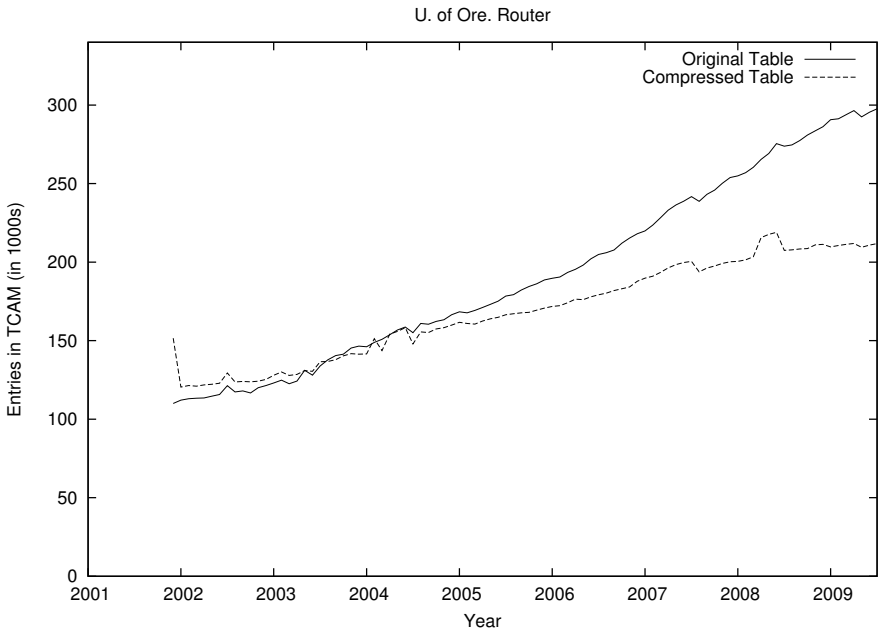

Fig. V.10. Scaling of Compression

E. Possible Enhancements

Routers often receive duplicate updates. Identifying such updates can reduce the processing time of the duplicate update, and the worst case delay in the processing of subsequent updates. To reduce processing time for duplicate updates, a cache of previous requests in the form of a hash table was implemented. This improved the worst case processing time for an update. But since every request received has to be looked up in the cache and then followed by a lookup in the tree, the average processing time of an update increased. The data reported in Table V.1 is from an implementation that does not utilize a cache of previous updates. Our implementation is easily amenable to the use of such a cache. If a small hardware cache of about a few KBytes were available, duplicate updates can be identified much faster, thereby reducing the worst case delay.

CHAPTER VI

CONCLUSIONS

To reduce the complexity, power consumption and lookup time of large IP routers, it is desirable to reduce the size of a routing table. In this paper, we propose a linear time algorithm to compress a routing table. We discussed the flaws in existing IP routing table compression algorithms, and discussed our strategy, which avoids these flaws by construction. Our algorithm is capable of compressing a routing table by about 25%, and can also handle routing table updates in real time. The most important advantage of our approach is that a longest prefix match determination unit is not required, since the routes that are mapped to different next hops are guaranteed to be orthogonal. This results in faster IP lookup operations and lowers power consumption. On an average our algorithm compressed a routing table to 24.6% of its original size. Also experimental results show that our algorithm is capable of handling route flapping and interface flapping in a real-time manner.

REFERENCES

[1] Y. Rekhter and T. Li, "An architecture for IP address allocation with CIDR," *RFC 1518*, 1993.

[2] V. Fuller, T. Li, J. Yu, and K. Varadhan, "Classless Inter-Domain Routing (CIDR): an address assignment and aggregation strategy," *RFC 1519*, 1993.

[3] Wikipedia, *Growth of global routing table*. http://en.wikipedia.org/wiki/border_gateway_protocol, Sep 2009.

[4] I. E. T. F. documents, *BGP Reports*. http://bgp.potaroo.net/.

[5] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high-speed IP routing lookups," *Computer Comm. Rev.*, vol. 27, pp. 25–36, Oct 1997.

[6] S. Sikka and G. Varghese, "Memory-efficient state lookups with fast updates," in *SIGCOMM '00: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Aug. 2000, pp. 335–347.

[7] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Trans. Computer Systems*, vol. 17, pp. 1–40, Oct 1999.

[8] I. Lee, K. Park, Y. Choi, and S. K. Chung, "A simple and scalable algorithm for the IP address lookup problem," *Fundam. Inf.*, vol. 56, no. 1,2, pp. 181–190, 2002.

[9] H. Lu and S. Sahni, "Enhanced interval trees for dynamic IP router-tables," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1615–1628, 2004.

[10] M. J. Akhbarizadeh and M. Nourani, "Hardware-based IP routing using partitioned lookup table," *IEEE/ACM Trans. Netw.*, vol. 13, no. 4, pp. 769–781, 2005.

[11] Z. Dai and B. Liu, "A TCAM based routing lookup system," in *ICCC '02: Proceedings of the 15th International Conference on Computer Communication*, Washington, DC, USA, 2002, pp. 1090–1096.

[12] S. Kumar, M. Becchi, P. Crowley, and J. Turner, "CAMP: fast and efficient IP lookup architecture," in *ANCS '06: Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, New York, 2006, pp. 51–60.

[13] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," *Proc. IEEE INFOCOM*, vol. 3, pp. 1240–1247, 1998.

[14] A. J. McAuley and P. Francis, "Fast routing table lookup using CAMs," *Proc. IEEE INFOCOM*, pp. 1382–1391, March-April 1993.

[15] T. B. Pei and C. Zukowski, "VLSI Implementation of routing tables: tries and CAMs," *Proc. IEEE INFOCOM*, vol. 2, pp. 515–524, 1991.

[16] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*, 2nd ed. Location: Prentice Hall, 2003.

[17] M. Kobayashi, T. Murase, and A. Kuriyama, "A longest prefix match search engine for multi-gigabit IP processing," *Proc. IEEE ICC*, vol. 3, pp. 1360–1364, June 2000.

[18] D. Shah and P. Gupta, "Fast updating algorithms for TCAMs," *Micro IEEE*, vol. 21, pp. 36–47, Jan/Feb 2001.

[19] H. Liu, "Reducing routing table size using ternary-CAM," *Hot Interconnects 9*, pp. 69–73, Aug. 2001.

[20] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Location: Kluwer Academic Publishers, 1984.

[21] R. P. Draves, S. Venkatachary, and B. D. Zill, "Constructing optimal IP routing tables," in *In Proc. IEEE INFOCOM*, 1999, pp. 88–97.

[22] J. Bian and S. P. Khatri, "IP routing table compression using ESPRESSO-MV," *ICON2003 IEEE Networks*, pp. 167–172, Oct 2003.

[23] R. Rudell and A. Sangiovanni-Vincentelli, "Espresso-MV: algorithms for multiple-valued logic minimization," in *Proceedings of the IEEE 1985 Custom Integrated Circuits Conference*, May 1985, pp. 230–4.

[24] V. C. Ravikumar and R. N. Mahapatra, "TCAM architecture for IP lookup using prefix properties," *IEEE Micro*, vol. 24, no. 2, pp. 60–69, 2004.

[25] University of Oregon Route Views Project, *BGP and FIB data*. http://www.routeviews.org.

[26] B. Gamache, Z. Pfeffer, and S. P. Khatri, "A fast ternary cam design for ip networking applications," in *Proc. the 12th International Conference on Computer Communications and Networks*, Oct. 2003, pp. 434–439.

VITA

Kalyana Chakravarthy Bollapalli received his B. Tech degree in electrical engineering from the Indian Institute of Technology (IIT) Bombay, India in 2004. His undergraduate research focused on performance estimation of Field Programmable Gate Array based fault simulator. After which he worked as an engineer in the automotive microcontrollers division of Infineon Technologies till 2007. There he worked on modeling and performance estimation of Infineon's 32-bit automotive controllers. In August of 2007 he joined Texas A&M University to pursue his master's degree in computer engineering and graduated in 2009. His current research interests include low power methodologies, circuit design, logic minimization and algorithms for VLSI CAD.

Publications:

- "*Highly Parallel Decoding of Space-Time Codes on Graphics Processing Units*," Bollapalli and Wu and Gulati and Khatri and Calderbank. Allerton 2009.
- "*A PLL Design based on a Standing Wave Resonant Oscillator*," Karkala and Bollapalli and Garg and Khatri. ICCD 2009.
- "*A Robust Pulse-triggered Flip-Flop based Enhanced Scan Cell Design*," Kumar and Bollapalli and Garg and Soni and Khatri. ICCD 2009.

Permanent Address:
332F WERC,
Texas A&M University,
College Station, TX-77843
e-mail: kalyan.bollapalli@gmail.com