A Memory-Efficient TCAM Coprocessor for IPv4/IPv6 Routing Table Update

Fang-Chen Kuo, Yeim-Kuan Chang, and Cheng-Chien Su

Abstract—Ternary content-addressable memory (TCAM) is a simple hardware device for fast IP lookups that can perform a lookup per cycle. However, prefixes may be inserted into or deleted from the TCAM because of changes in Internet topology. Traditional TCAM coprocessors maintain the enclosure relationship among prefixes by using an extended binary trie and perform TCAM movements based on an update algorithm (e.g., CAO_OPT) which runs on a local CPU to maintain the speed and correctness of the TCAM search process. In this paper, we propose a memory-efficient TCAM coprocessor architecture for updates that require only small memory size compared with the extended binary trie. The average number of TCAM movements per update is almost the same as that of CAO_OPT. However, the time to compute how to move TCAM entries in the proposed TCAM coprocessor is less than that in CAO_OPT. Only a small part of total TCAM search cycles is used to complete our update process. The proposed TCAM architecture can also be made smaller and faster because large off-chip memory for the extended binary trie and a local CPU are no longer necessary.

Index Terms—TCAM, IP lookup, longest prefix match, update

1 Introduction

The primary function of routers on the Internet is to forward IP packets to their final destinations [8]. Therefore, routers choose the next output port to send out every incoming packet. The rapid increase of Internet users causes IPv4 address exhaustion, which can be solved in the short term by classless interdomain routing [1]. Each routing entry uses a <prefix, prefix length> pair to increase the usage of IP addresses and longest prefix match (LPM) for routing table lookups. The LPM limitation complicates the routing table design. A long-term solution for insufficient IP addresses is IPv6 protocol defined by the Internet Engineering Task Force [2]. The IPv6 address length is 128 bits; thus, IPv6 is exhausted only after a long time.

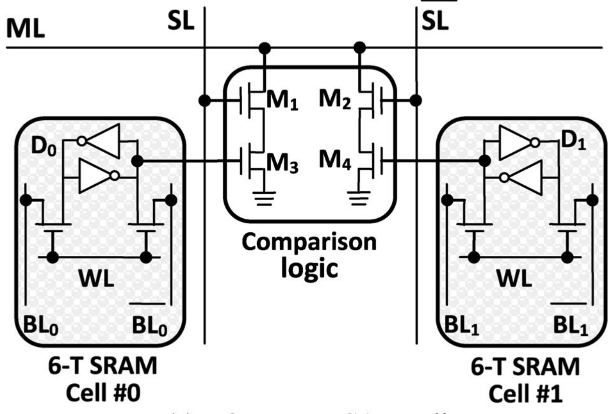
The packet forwarding design performs poorly at present. Each router searches its routing table by using destination IP addresses and forwards the packet to the next router based on the next port number. Therefore, a fast routing lookup scheme should be developed.

Ternary content-addressable memory (TCAM) devices are used widely as search engines for packet classification and forwarding in commercial routers and network devices [3], [4]. TCAM takes one cycle to search all routing entries in parallel at a very high lookup speed. For IP lookups, several prefixes (up to 32 for IPv4 and 128 for IPv6) may match the destination address of the incoming packet. However, only the longest prefix match is output. In contrast, conventional network processor-based software and field-programmable gate array/application-specific integrated circuit–based hardware designs that use various data structures may require multiple memory access for a single lookup.

In TCAM, each cell can be in one of three logic states: 0, 1, or * ("don't care" state). Fig. 1a shows the typical design of the NOR-type TCAM cell [12], in which two 1-bit 6-T static random-access memory (SRAM) cells (D_0 and D_1) are used to store the three logic states of a TCAM cell. Generally, the 0, 1, and * states of a TCAM cell are set by $D_0 = 0$ and $D_1 = 1, D_0 = 1$ and $D_1 = 0$, and $D_0 = 1$ and $D_1 = 1$, respectively (Fig. 1b). Each SRAM cell consists of two cross-coupled inverters (each formed by two transistors) and two additional transistors used to access each SRAM cell via two bitlines (BLs) and one wordline (WL) for read and write operations. Therefore, each SRAM cell has six transistors, and each TCAM cell consists of 16. The pair of transistors ($\mathrm{M}_1/\mathrm{M}_3$ or $\mathrm{M}_2/\mathrm{M}_4$) forms a pulldown path from the matchline (ML). If one pulldown path connects the ML to ground, the state of the ML becomes 0. A pulldown path connects the ML to ground when the searchline (SL) and D_0 do not match. No pulldown path that connects the ML to ground exists when the SL and D_0 match. When the TCAM cell is in the "don't care" state, M_3 and M_4 prevent searchlines SL and $\overline{SL}$, respectively, from being connected to ground regardless of the search bit. Similarly, when the search bit is "don't care" because $SL = 0$ and $\overline{SL} = 0$, M_1 and M_2 prevent searchlines SL and $\overline{SL}$, respectively, from being connected to ground regardless of the stored value in the TCAM cell. Fig. 1c shows the truth table of match line of a TCAM cell with inputs of search bits and the cell states. Multiple cells can be connected serially to form a TCAM word by shortening the MLs of adjacent cells. A match condition on the ML of a given TCAM word results only when the ML of every individual cell is not connected to ground. Several TCAM words can be connected by shortening all SLs to

• *The authors are with the Department of Computer Science and Information Engineering National Cheng Kung University, No. 1, Ta-Hsueh Road, Tainan 701, Taiwan, ROC.*
E-mail: {p7895107, ykchang, p7894104}@mail.ncku.edu.tw.

Manuscript received 20 Sep. 2012; revised 6 Jan. 2013; accepted 8 Jan. 2013. Date of publication 27 Jan. 2013; date of current version 7 Aug. 2014.
Recommended for acceptance by P. Bellavista.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.22



(a) NOR-type TCAM cell.

SRAM cell		TCAM state
D_0	D_1	
0	1	0
1	0	1
1	1	X
0	0	N/A

Search line		Search bit
SL	$\overline{SL}$	
0	1	0
1	0	1
0	0	X
1	1	N/A

(b) Ternary encodings for TCAM states and search bits.

TCAM state	Search bit		
	0	1	X
0	1	0	1
1	0	1	1
X	1	1	1

(c) Truth table for the matchline (ML).

Fig. 1. Typical TCAM device.

form a typical TCAM unit (Fig. 2). A logic unit called *priority encoder* is designed to output the index of the first matching TCAM word at the lowest (or upper) address. The ML sense amplifier (MLSA) between each TCAM word and priority encoder detects the state of its ML (high = match; low = miss).

Given that the search bit can be set to "don't care", a control register called *global mask register* can be used to mask TCAM columns that should not participate in the matching process. Any column can be masked by setting the corresponding global mask bit to 0, because both *SL* and $\overline{SL}$ are set to 0 via the bitline control logic. A TCAM search operation begins by first loading the search key and global mask into corresponding registers. Next, all MLs are placed temporarily in the match state by precharging them to high. Then, SL drivers broadcast differential searchlines (*SL* and $\overline{SL}$), and each TCAM cell compares the stored bit against the bit on its corresponding SLs. MLs which have all matching bits remain in a high state and the ones with at least one bit that misses discharge to ground. Finally, the encoder maps the ML of the matching location to its encoded address.

When global mask is used, match operations in TCAM become rich. In normal IP search mode, all global mask bits

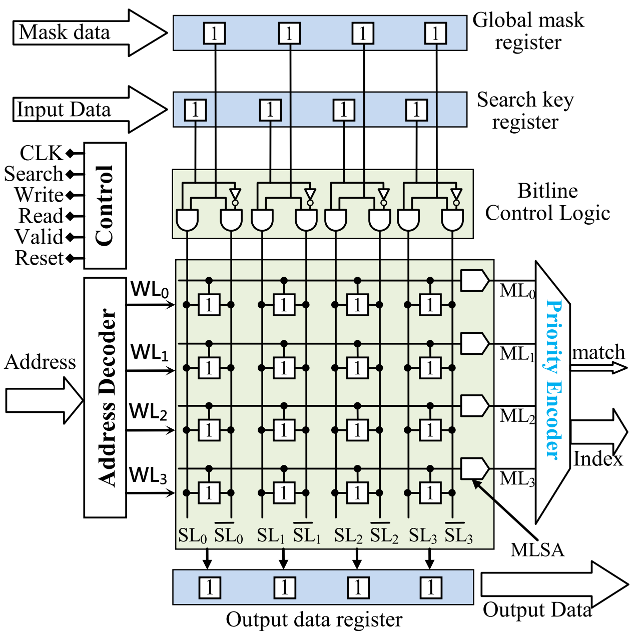


Fig. 2. Typical TCAM device.

are enabled. Fig. 2 shows a typical TCAM device that has four 4-bit prefixes. TCAM can be searched by using SLs and MLs. Like normal RAM, TCAM can also be read or written by using BLs and WLs. The input global mask is 1111, and the input search data is 1111. Two prefixes are matched against the search key 1111, and the priority selector returns the index 0 of prefix 1111. In the prefix search mode, a prefix can be used as the search key for a match in TCAM. For example, when the prefix of length 3, 110X, is used as the search key, the most significant three bits of the *global mask register* are enabled, and the fourth mask bit is disabled. Thus, only the most significant three bits of all TCAM entries are compared with 110. As a result, the search result returned by TCAM is the LPM that covers or is covered by the input prefix, which is 11^{**} in this example.

However, TCAM has some disadvantages, including high power consumption due to the charging and discharging processes of all TCAM cells, inefficiency of storing ranges due to the nature of ternary states in TCAM cells, and slow update operations. In this paper, we focus only on the TCAM update problem. Solutions for the other two problems can be found in [12], [21], [22], [23]. Typically, the prefixes stored in TCAM must be sorted in decreasing order of their lengths and so that the first matching prefix is the LPM.

Routing table update is traditionally performed by a local CPU via a CPU/TCAM coprocessor interface, such as the 64-bit PCI bus in an Intel IXP2800 network processor [17]. Fig. 3a shows the typical TCAM coprocessor architecture along with a network processor. Search requests are issued by lookup threads of the network processor, and update requests are issued by the local CPU. When TCAM content needs to be updated, the CPU locks the interface to avoid inconsistent and/or erroneous results. However, the disadvantage of interface locking is that all lookup threads must be suspended until the update process is complete, thereby impairing

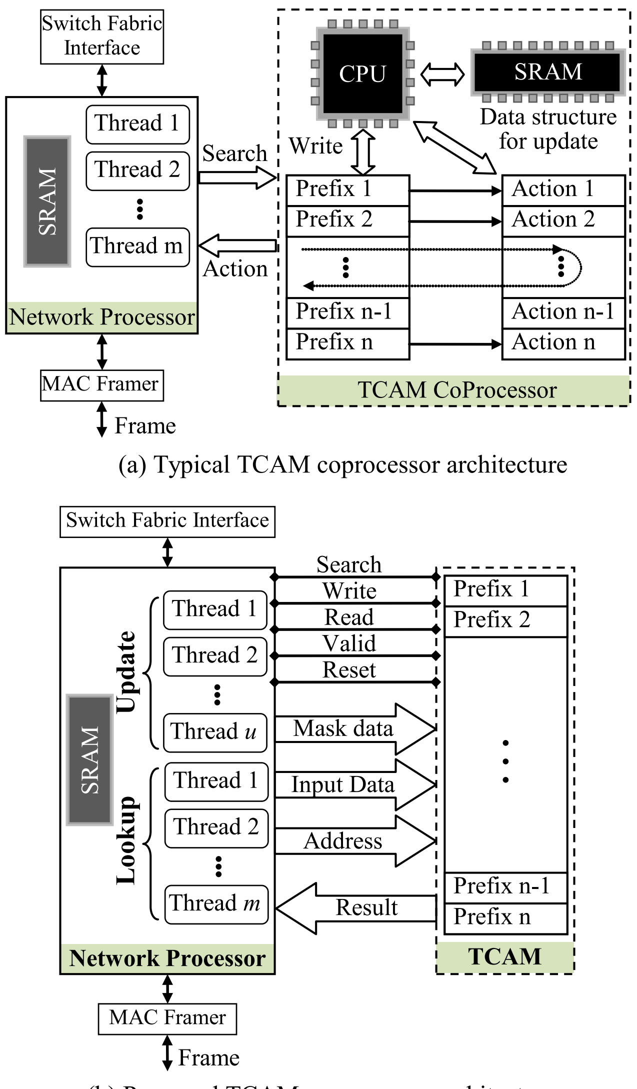


Fig. 3. TCAM coprocessor architecture. Typically, the search, read, and write operations in TCAM coprocessor take 1, 1, and 3 cycles, respectively.

overall search performance. This paper does not consider dual-port TCAM, which permits the local CPU and network processors concurrent access. Therefore, a non-blocking update algorithm should be designed. We also need a data structure usually stored in off-chip SRAM to compute how to perform the TCAM update process (Fig. 3a). In other words, we must know which TCAM entry needs modification or transfer to another TCAM entry to fulfill the update request. These update computations are performed by the CPU upon receiving an update request. One disadvantage is that the required data structure is usually large, such as the auxiliary binary trie proposed in [6].

In this paper, we propose a memory-efficient TCAM update architecture for IPv4/IPv6 routing lookups (Fig. 3b). The new update scheme does not require large memory for the extended binary trie data structure in existing schemes [6], [9], [10], [14], [15]. Thus, off-chip memory and local CPU are not necessary in the TCAM coprocessor. Computing how to modify TCAM content can be made faster than the computation based on the extended binary trie. The proposed update process performed by update threads in the network processor is also non-blocking.

We use the network processor Intel IXP2800 as an example to explain how the proposed update scheme can be integrated into routers. In IXP2800, the packet processing procedure follows the pipeline model which has three stages: receive, process, and transmit. Several threads, called receive, lookup, and transmit threads, are allocated independently to these stages. For packets that take longer processing time, more threads are usually allocated, or the procedure becomes stuck and packets are dropped. A global FIFO queue can be used for load balancing between different stages. That is, one queue can be allocated between the receive and process stages (Queue A), and another between the process and transmit stages (Queue B). The basic processing flow works as follows: One receive thread enqueues the packet in Queue A after receiving it. Then, the lookup thread dequeues the packet from Queue A and enqueues the packet in Queue B after searching TCAM to obtain next hop information. Finally, transmit threads dequeue the packet from Queue B and forward it to the correct next hop.

We can allocate unused threads for the update to realize the proposed TCAM coprocessor design. We allocate another queue (Queue C) to buffer packets that result in prefix updates. If a lookup thread finds that the packet dequeued from Queue A is for update, it enqueues the packet in Queue C, rather than in Queue B. After dequeuing the packet from Queue C, update threads handle the necessary update operations. As a result, updates can be handled by the network processor without the help of local CPU.

The rest of the paper is organized as follows. Section 2 provides background knowledge on the topic. In Section 3, we analyze the prefix enclosure relationship and propose a memory-efficient TCAM update scheme based on this analysis. Performance comparisons are presented in Section 4, and concluding remarks are given in the last section.

2 Related Work

Many TCAM update algorithms have already been proposed; this section presents a survey of these algorithms. The TCAM update process consists of three major parts. In the first part, we determine the TCAM entry where a prefix can be stored so that the TCAM device can return the LPM to the destination IP address of the incoming packet. Two well-known constraints, namely, prefix-length ordering (PLO) and chain-ancestor ordering (CAO), were proposed for this purpose [6]. After the prefix ordering constraint is determined, we have to compute prefixes that should be moved to other positions after a prefix is inserted or deleted to maintain correct prefix ordering. Based on the ordering constraint, a prefix can be moved to more than one position. Normally, this step cannot be done without a data structure (e.g., a binary trie) that keeps track of prefix storage in TCAM. In the third part, we move the prefixes determined by the second part from one position to another. The efficiency of this step may depend on the location or pool of unused entries.

Typically, prefix order should follow the PLO constraint [6]. According to the PLO constraint, two prefixes of the same length do not require a special order because they are disjoint; thus, only one of them may match the search key. On the other hand, if more than one prefix matches the search key, one must enclose the other; thus, their lengths must be different. According to this rule, only prefixes of different lengths require sorting. As a result, the naïve method for allocating prefixes in TCAM is to divide prefixes into groups according to length. A maximum of W groups are allocated to the W-bit IP address space (e.g., 32/128 for IPv4/IPv6). All groups are then stored in TCAM in decreasing order of their lengths. The free pool of unused entries is placed at the bottom of the TCAM.

With the PLO constraint, a mirrored copy of all TCAM entries in a linear array can serve as the data structure for monitoring the prefixes stored in TCAM. This linear array is augmented with boundary indices to record the start and finish prefixes of all groups divided by their lengths. Insertion is simple because the length of the prefix to be inserted is known. However, when deleting a prefix P, the TCAM coprocessor has to search the linear array in off-chip memory to check the existence of P in TCAM and the TCAM entry that stores P. The worst-case number of memory movements for this method is $O(W)$.

The PLO_OPT algorithm proposed in [6] also joins prefixes of the same length. However, the free pool is placed in the center of the TCAM (i.e., between the groups of lengths $W/2-1$ and $W/2$). Prefixes with length W to $W/2$ are stored above the free pool, and prefixes with length $W/2-1$ to 1 are stored below the free space. The update algorithm is the same as the naïve method. The worst-case number of memory movements is $O(W/2)$.

Although the PLO update algorithm is simple, the number of TCAM entry movements is much higher than that of the CAO constraint also proposed in [6]. As stated above, if two prefixes match the search key, one must enclose the other. All prefixes with this enclosure relationship form a prefix enclosure chain. Therefore, only prefixes in a prefix enclosure chain have to be arranged in decreasing order of their lengths. In CAO_OPT, free space is also maintained in the center of TCAM. Every prefix enclosure chain is cut in half by the free space. Suppose D is the length of a chain in the trie structure. The distance from every prefix in this chain to the free space is less than $D/2$. Thus, after the update, the number of TCAM entry movements in this chain is less than $D/2$, where D is the maximum length of the chain. In practice, $D = 7$ in IPv4, and $D = 11$ in IPv6.

The augmented binary trie proposed in CAO_OPT [6] for monitoring prefixes stored in TCAM is more complex than the linear array required in PLO. The size of off-chip memory for the binary trie is restricted when IPv6 is considered. Aside from left and right child pointers, some fields that require pre-computation are *wt*, *wt_ptr*, and *hcld_ptr* for each node in the binary trie [6]. Assume that each pointer requires four bytes and that *wt* requires 1 byte. Each binary trie node takes 17 bytes. For the IPv4 routing table AS6447 (2011-04-20), which consists of 367,719 prefixes, the binary trie consists of 907,221 nodes. Thus, 15 MB memory is required, which can be placed only in off-chip memory. When a prefix p is inserted or deleted, the following information has to be computed. 1) the longest chain which comprises p denoted by $LC(p)$ and its length (i.e., the number of prefixes in the chain), 2) the path from the trie root to node p, 3) a child node of p with a prefix, 4) the highest prefix-child of p denoted by $hcld(p)$ in terms of the TCAM location of this child, and 5) the chain denoted by $HCN(p)$ which comprises ancestors of p, prefix p itself, $hcld(p)$, $hcld(hcld(p))$, and so on. As analyzed in [6], $LC(p)$ and $HCN(p)$ are determined in $O(W)$ time. In sum, the two disadvantages are: 1) computing information for CAO is time consuming, and 2) storing this information requires large off-chip memory. The proposed TCAM update scheme aims to solve these two disadvantages.

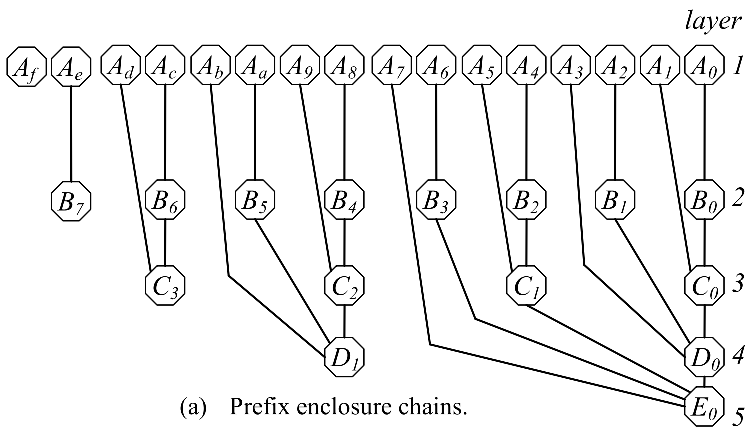
TCAM update schemes for packet classification are proposed by [18], [19], [20]. In [18], authors propose a consistent algorithm for updating a batch of rules. This algorithm focuses on how to maintain the consistency of a filter set to avoid locking the TCAM coprocessor during the update process. In [19], authors propose a fast way to update rules by inserting new filters into arbitrary entries without considering the priority order among overlapped filters. However, this approach slows the lookup speed because it requires $\log_2 N$ lookups, where N is the number of the distinct priorities of rules, to determine the filter with the highest priority. In [20], authors propose the use of two kinds of TCAM (i.e., LTCAM and ITCAM) to store filters. LTCAM stores filters with the highest priority; all filters in LTCAM are disjoint. Remaining filters are stored in ITCAM, and the priority graph proposed by [18] is required to manage the filters in ITCAM to ensure correct lookups. A multidimensional trie is created to determine the filters to be stored in LTCAM. LTCAM and ITCAM are searched in parallel. The search result of ITCAM is discarded if a match is found in LTCAM because the matched filter in LTCAM must have a higher priority than the one in ITCAM.

3 Proposed TCAM Update Scheme

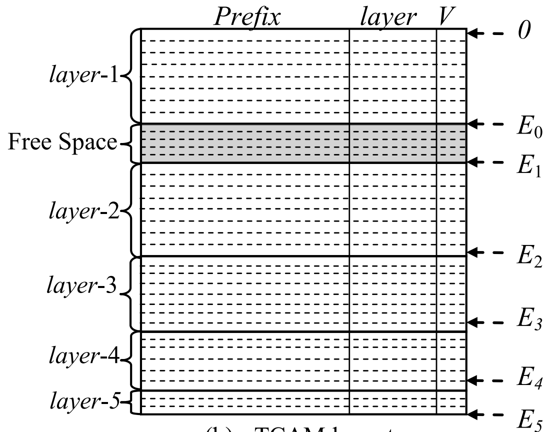
In this section, we describe some notations and prefix properties needed in this paper. Then, the prefix enclosure relationship is analyzed for routing tables available from typical routers on the Internet. Based on the analyzed results, a novel memory-efficient TCAM update scheme is proposed.

A W-bit prefix P of length *len* is normally denoted as $b_{W-1}\ldots b_{W-len} * \ldots *$ ($W - len$ trailing don't cares) in ternary format, where $b_i = 0$ or 1 for $i = W-1$ to $W - len$. Prefix P of length *len* is a range of consecutive addresses from $b_{W-1}\ldots b_{W-len}0\ldots 0$ to $b_{W-1}\ldots b_{W-len}1\ldots 1$. A prefix is said to be enclosed or covered by another prefix if the address space covered by the former is a subset of that covered by the latter. For two distinct prefixes A and $B, B \supset A$ or $A \subset B$ is used to denote that B encloses A, where $\supset$ or $\subset$ is called the enclosure operator. Two prefixes are said to be disjoint if none of them are enclosed by the other.

The de facto standard inter-domain routing protocol on the Internet is the border gateway protocol (BGP), which provides loop-free routing between autonomous systems (ASs) [13]. Each AS independently administers a set of routers. The address space represented by an advertised


layer
A_f A_e A_d A_c A_b A_a A_9 A_8 A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0 1
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0 2
C_3 C_2 C_1 C_0 3
D_1 D_0 4
E_0 5

(a) Prefix enclosure chains.



(b) TCAM layout.

Fig. 4. Sample routing table of five layers.

BGP prefix may be a sub-block of another existing prefix. The former is called *covered* prefix and the latter is called *covering* prefix. Covered and covering prefixes form a prefix enclosure chain.

We analyze three BGP IPv6 routing tables from [5] to determine the detailed enclosure relationship between covered and covering prefixes. Given that these existing IPv6 tables are too small compared with IPv4 tables, we also generate two routing tables larger than 120,000 prefixes (V6Gene1 and V6Gene2) using the generator in [7].

In theory, one IPv6 prefix may be covered by at most 127 prefixes. Therefore, the worst-case enclosure chain size is 128 for IPv6. To precisely describe the prefix enclosure relationship, the prefixes are grouped into *layers* based on the following *layer grouping scheme*: Prefixes that do not cover any other prefix in the routing table are grouped into layer 1. Prefixes that cover only layer 1 prefixes are grouped into layer 2 and so on.

Definition 1. *P_k denotes a prefix in layer k.*

Definition 2. *Parent (P) denotes the immediate parent prefix of P on an enclosure chain comprising P.*

Fig. 4a shows the prefix enclosure relationship for a routing table with five layers (i.e., the maximum enclosure chain size is 5) and the number of prefixes in each layer in Table 1 for all five routing tables. Layer 1 prefixes account for 92 to 96 percent of prefixes in the routing table. Layer 2 prefixes account for 2 to 6 percent of the prefixes. Prefixes in other layers account for only less than 1 percent of total prefixes. Some important properties of the routing tables based on the prefix enclosure relationship are summarized below. A layer-$(i+1)$ prefix P_{i+1} covers at least one layer-i prefix P_i. In other words, P_{i+1} must be *parent*(P_i). However, if a layer-j, P_j, is parent$(P_i), P_j$ can be in any layer j, where $j \geq i+1$:

1. A layer-$(i+1)$ prefix P_{i+1} covers at least one layer-i prefix P_i. In other words, P_{i+1} must be *parent* (P_i). However, if a layer-j P_j is parent(P_i), P_j can be in any layer j, where $j \geq i+1$.
2. If $P_i \supset P$ and $P_j \supset P$, we must have $P_j \supset P_i$ for $j > i$. P_j and P_i must also be in the same enclosure chain of P.
3. Prefixes in the same layer are disjoint.
4. For two prefixes P_i in layer-i and P_j in layer-j, $P_j \supset P_i$ is not always true.
5. For any prefix P_j in layer-j, a prefix P_i must exist in layer-i for $i = 1$ to $j-1$ such that $P_j \supset P_{j-1} \supset \cdots \supset P_2 \supset P_1$.
6. If an input IP address has multiple matches in different layers, the matched prefix in layer i must be longer than that in layer j for $i < j$.

According to our analysis of various routing tables, layers 1 and 2 have 98 to 99 percent of the prefixes.

As for properties 1 and 2, we propose to divide TCAM into L + 1 layers for routing tables with L layers. The additional layer corresponds to the free TCAM space. Based on property 2 of the routing tables, layer i is placed above layer j for $i < j$. As a result, if multiple matches are found against the input IP address, the priority encoder can return the LPM correctly. Given that the majority of prefixes are in layers 1 and 2, most updates must be allocated for the prefixes in these two layers. Thus, the average number of TCAM entry movements can be reduced by placing the free space between layers 1 and 2. Fig. 4b shows the proposed TCAM layout for the routing table of five layers. For the routing table of L layers, an index array of size $L+1, E_i$ for $i = 0$ to L, is used in the proposed TCAM update

TABLE 1
Routing Table Grouping Results

Database	V6Gene1	V6Gene2	AS2.0	AS1221	AS6447
Size	120,861	127,697	3,001	933	3,090
Layer-1	115,709 (95.7%)	123,279 (96.5%)	2,808 (93.6%)	867 (92.9%)	2,870 (92.9%)
Layer-2	2,875 (2.4%)	2,770 (2.1%)	184 (6.1%)	62 (6.6%)	205 (6.6%)
Layer-3	1048 (0.9%)	794 (0.6%)	8 (0.3%)	3 (0.3%)	13 (0.4%)
Layer-4	563 (0.5%)	428 (0.3%)	1	1	2
Layer-5	269 (0.2%)	183 (0.1%)			
Layer-6	169 (0.1%)	87			
Layer-7	130	71			
Layer-8	57	39			
Layer-9	23	27			
Layer-10	12	11			
Layer-11	6	8			

algorithms, as follows. E_i records the index of the lowest entry with the lowest memory location in layer i for $i = 2$ to L, E_1 records the index of the lowest entry in the free pool of unused TCAM entries, and E_0 records the index of the lowest entry in layer 1. The W-bit prefix is stored in the *prefix field* of TCAM entries, where W is 32 and 128 for IPv4 and IPv6, respectively. A *layer field* of k additional bits is also required to store the layer number for each prefix for the proposed TCAM update scheme. Given that a maximum of 11 layers are analyzed for the routing tables, $k = 4$ is sufficient. Finally, a *valid bit field* in the TCAM is used to activate a TCAM entry in the search process if the corresponding valid bit is enabled.

According to the proposed layered scheme, the CAO constraint is preserved. No augmented binary trie is required to maintain the CAO. Below, we discuss the maintenance scheme of the proposed scheme when a prefix is inserted or deleted.

P is the prefix to be inserted. The enclosure relationship between P and the prefixes in the routing table must first be determined. As stated in the layer grouping scheme, if P does not enclose any prefix in the routing table, it must be inserted in layer 1. However, if a layer 1 prefix Q covers P, Q must be moved up to layer 2. If Q is also covered by another layer 2 prefix R, R must be moved up to layer 3. In general, if P is determined to be inserted in layer i, whether or not a prefix Q is present in layer i that covers P must be checked. If so, then Q will be inserted in layer $i + 1$ before P is inserted in layer i. Specifically, P is written in the same entry as Q's previous location in layer-i.

The operation of deleting an existing prefix P must now be considered. If P is in layer i, the layer-$(i + 1)$ prefix Q that covers P must be determined. If Q does not exist, P can be deleted directly. Otherwise, whether or not Q also covers another layer-i prefix (say R) must be checked. If R exists, P can be deleted directly. Otherwise, if Q does not cover at least two prefixes (say P and R), the enclosure chain relationship will fail to maintain after P is removed. Thus, Q has to be moved down to layer i. This case is implemented in two steps: Q overwrites P in layer i, and deletes Q in layer $i + 1$. These two steps are performed recursively until no layer movement is needed. The prefix movements needed to maintain the chain-ancestor order for insertion or deletion are called *enclosure chain prefix movement*.

To avoid performing TCAM entry movements too many times, a free list for each layer is chosen to record the positions of the unused TCAM entries. The free list of layer i is denoted by *FreeList*[i] which can be implemented by a linked list. Since layers 1 and 2 are next to the free prefix pool, their free lists are not empty all the time.

As described, the TCAM updates primarily decide which TCAM entries should be moved to other positions for maintaining the prefix enclosure chain order of the layer grouping scheme. A special data structure is needed to quickly determine which prefixes are affected by the newly inserted or deleted prefix. This special data structure is usually large and stored in off-chip memory which can be accessed by a TCAM coprocessor.

Subsequently, a novel TCAM entry movement decision scheme is proposed, which does not need a large amount of the off-chip memory for the augmented binary trie in CAO or for the linear array in PLO. In addition to the arrays E_i and *FreeList*[i], only a small amount of memory (359 KB for 367,719 prefixes) is needed to record the length of each prefix denoted by *SRAM*[i]*.len* for $i = 1$ to N, where N is the number of entries in TCAM. As stated, the extra free bits that are left unused in TCAM entries are used as the *layer field* to store the layer number of each prefix. By using the layer number, the TCAM entries that need to be moved when a prefix is inserted or deleted can be determined.

```
// k is the number of extra bits for layer field.
// '+' is the string concatenation operator.
// TCAM width is 144 bits and IP address is W bits wide.
LookupTCAM(IP, len, h)
{
01  for (i=0 to len–1) GlobalMask[i] = 1;
02  for (i= len to 143) GlobalMask[i] = 0;
03  if (h > 0) for (i= W to W+k–1) GlobalMask[i] = 1;
04  InputKey = IP + Binary(h);
05  result = TCAMSearch(InputKey, GlobalMask);
}
```

Fig. 5. Lookup TCAM.

The prefix field in TCAM is 128 bits denoted by $b_{0,} \ldots, b_{127}$ and the layer field is $b_{128,} \ldots, b_{131}$. As analyzed previously, a maximum of 11 layers is needed for the IPv6 routing tables used in this paper; thus, 4 extra bits are enough. Currently, TCAM devices can be configured to be 36, 72, 144, or 288 bits wide. Therefore, configuring TCAM entries into 144 bits wide is the most suitable for IPv6. As a result, 16 unused bits are available in each entry from which four bits can be used for storing the layer number of each prefix. These unused bits are used only for update operations. Therefore, when performing the lookup operations, setting the corresponding global mask bits to 0 masks the unused bits. Below, the TCAM updates are performed with the 4-bit layer field.

3.1 TCAM Search

This section shows the method used in performing the search operations in the TCAM with k extra bits for layer numbers appended to each TCAM entry. As stated above, the k extra bits are called the *layer field* to differentiate it from the prefix field in TCAM entries. Fig. 5 shows the detailed search algorithm *LookupTCAM*(*IP*, *len*, *h*). The pair of parameters *IP* (the W-bit IP address) and *len* is used to represent the prefix to be searched. Parameter h is designed to select the layer so that the matched prefix against search key can only be in layer h. If $h = 0$, the global mask of bits W to $W + k - 1$ for layer numbers will be disabled. Thus, all the prefixes will be matched as in the normal IP lookups. In line 4 of the algorithm, the search key Q is constructed to be the IP address appended with the binary string of layer number h. Finally, the TCAM query denoted by *TCAM-Query*(Q, *GlobalMask*) is executed by the TCAM coprocessor with the search key stored in Q and the search mask stored in *GlobalMask*. The result obtained from the TCAM coprocessor consists of two parts, *match* and *idx*. If *result.match* $= 1$, the TCAM entry at index *result.idx* (denoted by

TABLE 2
Results after Searching Layer h with Key P

Case	Subcase	Possible layer for P
result.match=0 no match	N/A	1 to h
result.match=1 match = P_h	$P \supset P_h$	h+1 to L+1
	$P = P_h$	P exists in layer h
	$P_h \supset P$	1 to h

TABLE 3
Results after Searching All Layers with Key P

Case	Subcase	Possible layer for P
result.match=0 no match	N/A	1
result.match=1 match = P_m	$P \supset P_m$	2 to L+1
	$P = P_m$	P exists in layer h
	$P_m \supset P$ (m>1)	1
	$P_m \supset P$ (m=1)	1

TCAM[*result.idx*]) matches the search key. Otherwise, *result. match* $= 0$ and *result.idx* is unused.

Given the search key $P = IP/len$, two kinds of search operations can be performed: one for searching the prefixes in a specified layer and the other for searching all the prefixes as follows.

3.1.1 Search Layer h

To search the prefixes in layer h, bits W to $W-k+1$ of global mask register are activated as shown in line 3 of Fig. 5. As a result, only the prefixes in layer h can match the search key. Since all the prefixes in one layer are disjoint, only one match exists in layer h. We analyze the two cases based on the search result for prefix P as follows. Table 2 summarizes the search results.

1. *result.match* $= 0$: Layer h contains no matched prefix against the search key P. In other words, P is disjoint from all the prefixes in layer h. P may be in layer h or lower (i.e., P cannot be in the layer higher than h) because of the following reasons. P is assumed to be in layer y, where $y > h$. Based on the prefix grouping scheme, P must also enclose a prefix P_k in layer k for $k = 1$ to $y-1$ along the prefix enclosure chain of P. Therefore, it is a contradiction because P is disjoint from all the prefixes in layer h. As a result, P may be in layer h or lower. P_h is a prefix in layer h and P_i is a descendant prefix on the prefix enclosure chain of P_h, where $i = 1$ to $h-1$. P must not be enclosed by P_i because otherwise, P will also be enclosed by P_h, which is a contradiction. Furthermore, P must not enclose P_i since it is not possible for a prefix P to exist such that $P \supset P_i$ and $P_h \supset P_i$ but not $P_h \supset P$.
2. *result.match* $= 1$: A matched prefix P_h stored in TCAM entry *TCAM[result.idx]* is found in layer h. To determine whether P_h encloses or is enclosed by P, we have to compare their prefix lengths, i.e., *SRAM*[*result.idx*].*len* and the input *len*. If *SRAM* [*result.idx*].*len* $=$ *len*, P already exists in layer h. If *SRAM*[*result.idx*].*len* $>$ *len*, P encloses the matched prefix in layer h (i.e., $P \supset P_h$). Based on the prefix grouping scheme, P must also enclose a prefix P_k in layer k for $k = 1$ to $h-1$. As a result, P must be placed in one of the layers from $h+1$ to $L+1$. To check if P encloses, is enclosed by, or is disjoint from P_k for $k = h+1$ to L, we have to search layers $h+1$ to L, where L denotes the number of layers contained in the routing table. If *SRAM*[*result.idx*]. *len* $<$ *len*, P is enclosed by the matched prefix P_h in layer h (i.e., $P_h \supset P$). When $P_h \supset P$, P can be in any layer from 1 to h. Let $Desc_i(P_h)$ be a descendant prefix on a prefix enclosure chain of P_h, where $i = 1$ to $h-1$, as shown in Fig. 4a. If we assume $h = 5$ and P_h is E_0, $Desc_i(E_0)$ could be one from the set $Covered\quad(E_0) = \{D_0, C_0, C_1, B_0 \ldots B_3, A_0 \ldots A_7\}$.The enclosure relationship between P and $Desc_i(P_h)$ is unknown and P must be disjoint from any prefix other than the ones in $Covered(E_0)$. If P encloses or is enclosed by any prefix in $Covered(E_0)$, some prefix movements among layers must be required.

Example 1. Layer 3 is searched with the search key $= P$ in Fig. 4a. When no match is found, P cannot enclose or be enclosed by any descendant prefix of C_0, such as A_0, B_0, A_4, B_2, etc. However, P can enclose, be enclosed by, or be disjoint from any other prefix in layers 1 and 2, such as A_2, A_3, B_1, B_3, etc. Now, if a matched prefix C_0 is found, then if P encloses $C_0(i.e., P \supset C_0), P$ also encloses prefixes B_0 and A_0 in Fig. 4. However, the enclosure relationship between P and D_0 or E_0 is unknown. Similarly, if C_0 encloses P (i.e., $C_0 \supset P), D_0$ and E_0 also enclose P. The enclosure relationship between P and A_0, A_1 or B_0 is unknown. Furthermore, P must be disjoint from any other prefixes other than A_0, A_1 and B_0.

3.1.2 Search All Layers

To search all the prefixes in the TCAM, the global mask bits that correspond to the k extra bits in the layer field of all TCAM entries are deactivated. If *result.match* $= 0$ is returned after the TCAM search operation, P must be disjoint from all the prefixes in TCAM. If P is the prefix to be inserted, P should clearly be inserted in layer 1. Otherwise, if *result.match* $= 1$ is returned, the layer in which the matched prefix locates can be determined by checking the array E_i. Assume that the matched prefix is the prefix P_m in layer m. Two subcases are analyzed: 1) $P = P_m$ or $P \supset P_m$ and 2) $P_m \supset P$ as follows. Table 3 summarizes the search results

1. $P = P_m$ *or* $P \supset P_m : P_m$ must be in layer 1 (i.e., $m = 1$) because of the following reason. Assume $m > 1$. Based on the layer grouping scheme, P_m must also enclose another prefix P_i in layer i for $i \in \{1 \ldots m-1\}$. This is a contradiction because the TCAM coprocessor should have returned P_1 as the longest prefix match. If $P = P_m, P$ is already in layer 1. However, if $P \supset P_m, P$ can be in layer 2 or higher.
2. $P_m \supset P$: Consider two subcases, $m > 1$ and $m = 1$. In the first subcase, P must be disjoint from all the prefixes covered by P_m in layers 1 to $m-1$. This result is obtained because if P encloses a prefix P_i in layer i for $i = 1$ to $m-1$, it must also enclose a prefix P_1 in layer 1 which should have been returned as P's

```
Disjoint_InsertTCAM(IP, len, h) //Insert a prefix into layer h
{
01 if (FreeList[h] ≠ NULL) {
02     Remove the first free index (idx) from FreeList[h];
03     TCAM_Write(idx, IP, len, h);
04     return;
05 }
06 Find the nearest layer u to layer h when FreeList[u]≠NULL;
07 Remove the first free index (idx) from FreeList[u];
08 if (u < h){
09     TCAM_Write(idx, TCAM[E_u].layer, SRAM[E_u].len, u);
10     Decrement E_u by one;
11     for (i = u + 1; i<h; i++) {
12         TCAM_Write(E_{i–1}+1, TCAM[E_i].layer, SRAM[E_i].len, i);
13         Decrement E_i by one; }
14     TCAM_Write(E_{h–1}+1, IP, len, h);
15     if (h = L+1){ E_{L+1} = E_L + 1; L = L + 1;} //Add a new layer
16 } else { // u > h
17     k = E_{u–1}+1;
18     TCAM_Write(idx, TCAM[k].layer, SRAM[k].len, u);
19     Increment E_{u–1} by one;
20     for (i = u – 1; i > h; i––) {
21         k = E_{i–1}+1;
22         TCAM_Write(E_i, TCAM[k].layer, SRAM[k].len, i);
23         Increment E_{i–1} by one;
24     }
25     TCAM_Write(E_h, IP, len, h);
26 }
}
```

Fig. 6. Algorithm to insert a prefix in layer h.

longest prefix match by the TCAM. Thus, P belongs to layer 1. After P is inserted, the layer numbers of all the existing prefixes will remain the same. In the second subcase, P is enclosed by a prefix P_1 in layer 1. Thus, P also belongs to layer 1. If P is the prefix to be inserted, P is placed in the layer, and all the prefixes that enclose P (i.e., the prefixes on the enclosure chain of P) must be shifted up by one layer according to the prefix grouping scheme.

3.2 TCAM Update

Inserting a prefix P involves determining which layer to store P and which prefixes should be moved to other layers after P is inserted. Similarly, deleting a prefix P involves determining which layer contains P and which prefixes should be moved to other layers after P is deleted. The prefix movements after a prefix is inserted or selected are needed because the correct prefix enclosure chain order needed in the layer grouping scheme must be maintained. The algorithms for inserting and deleting a prefix $P = IP/len$ involve two search operations. Layer h and all the layers must be searched by calling *LookupTCAM*(*IP*, *len*, $h > 0$) and *LookupTCAM*(*IP*, *len*, $h = 0$) respectively.

The TCAM is used to write operations to move the content of a TCAM entry from one place of the TCAM entry to another. When we are ready to move *TCAM*[*i*] to *TCAM*[*j*], *TCAM*[*j*] must be unused. Otherwise, *TCAM*[*j*] must be moved to another place to make *TCAM*[*j*] available. This TCAM entry moving process can be performed recursively. As a result, a series of TCAM writes may be required. Before describing how to perform the computations needed to determine where to insert P or where to delete P from, the process of inserting a prefix P in layer h or deleting a prefix P stored in layer h is shown as follows.

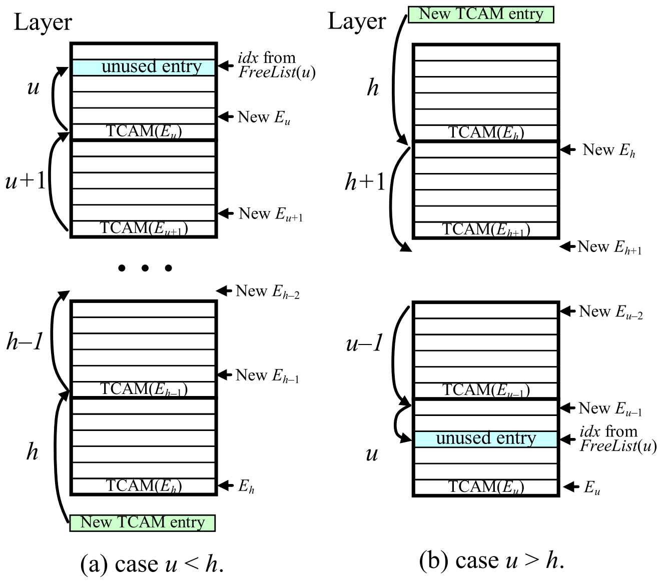


Fig. 7. Inserting a new TCAM entry in layer h.

3.2.1 *Insert a Prefix P in Layer h*

Two cases are considered. The first case assumes that P is disjoint from all the prefixes already in layer h. Since no free entry for P in layer h exists, some TCAM entry movements must be made to allocate a free entry. The second case assumes that layer h contains another prefix that is not disjoint from P; therefore, the enclosure chain prefix movement described earlier must be performed in this section.

P Is Disjoint from All the Prefixes in Layer h. Fig. 6 shows the detailed algorithm for the first case, *Disjoint_InsertTCAM*(*IP*, *len*, *h*). The free lists of layers 1 and 2 must not be empty because they are next to the free TCAM pool. In general, if the free list of layer h, *FreeList*[h], is not empty, an unused TCAM entry can obtain *FreeList* [h] to store P as shown in lines 1 to 5 of Fig. 6. If the free list of layer h is empty, the layer u is located with a non-empty free list closest to layer h. To do so, the array *FreeList* can be examined. The TCAM movements to create an empty slot in layer h are implemented in lines 8 to 26 where two cases $u < h$ and $u > h$ are considered separately. The case of $u < h$ is illustrated in Fig. 7a. An unused entry in layer $u + 1$ can be created by moving the border entry TCAM [E_u] of layer u to one of its free slots and decreasing E_u by one. This unused entry now becomes the uppermost entry of layer $u + 1$, denoted by TCAM [$E_u + 1$]. Similarly, the uppermost entry of layer i can be also made available by moving the entry TCAM [E_i] of layer i to TCAM [$E_{i-1} + 1$] and decreasing E_i by one repeatedly from $i = u + 2$ to $i = h$. Finally, the uppermost unused entry of layer h can be used to hold the newly inserted entry as shown in line 14. The case for $u > h$ is similar as shown in lines 16 to 26 of Fig. 6 and illustrated graphically in Fig. 7b.

Layer h Contains a Prefix that Is Not Disjoint from P. The detailed algorithm, *Chain_InsertTCAM*(*IP*, *len*, *h*), is

```
Chain_InsertTCAM(IP, len, h)
{
01 for (i = h; i <= L ; i++) {
02     if (SearchedResult[i] = NULL)
03         result = LookupTCAM(IP, len, i);
04     else result = SearchedResult[i];
05     if (result.match=0) {
06         Disjoint_InsertTCAM(IP, len, i); return;}
07     $P_i$ = IP/SRAM[result.idx].len; // The matched prefix
08     if ($P_i$ = P) return; //P exists
09     if ($P_i \supset P$){
10         TCAM_Write(result.idx, IP, len, i);
11         swap(SRAM[result.idx].len, len);
12     }
13 }//end for
14 Disjoint_InsertTCAM(IP, len, L+1);//create a new layer
}
```

Fig. 8. Insert prefix in layer h.

shown in Fig. 8 for the second case. The array *SearchedResult*$[i]$ for $i = 1, \ldots, L$ is used to record if layer i has already been searched during the process of determining in which layer P should be inserted. The searched result for a prefix P can be used for another prefix Q based on property 2 stated in Section 3. Thus, lines 2 to 4 of Fig. 8 verify if a TCAM lookup in layer i is needed based on the value of *SearchedResult*$[i]$. If the prefix P_i that encloses P exists in layer i, P is inserted in layer i in this iteration of the loop, and P_i will be inserted in layer $i{+}1$ in the next iteration, as shown in lines 7 to 12. Finally, if all the layers are exhausted, a new layer will be created and P is inserted in the new layer as shown in line 14.

3.2.2 Delete a Prefix P from Layer h

Consider deleting a TCAM entry (say TCAM[*idx*]) from layer h, where *idx* is determined in the first step. Free lists are used to record the positions of the deleted TCAM entries. Therefore, only the valid bit field of TCAM[*idx*] is set to 0. No physical TCAM movement is required, as shown in Fig. 9.

3.3 Prefix Insertion

Fig. 10 shows the detailed algorithm *Insert*(*IP*, *len*, *L*) to insert prefix $P = IP/len$ into the TCAM containing L layers. This algorithm takes $O(\log L)$ steps to determine the layer in which the new prefix should be put. The algorithm starts with a lookup on all the prefixes in the TCAM coprocessor by calling *LookupTCAM*(*IP*, *len*, *0*). If no match is returned from TCAM coprocessor, P must be disjoint from all the prefixes; thus, it is inserted in

```
// Delete the entry TCAM[idx] in layer h
DeleteTCAM(idx, h)
{
01     TCAM[idx].V = 0; //disable the valid bit of the entry
02     Store index idx in FreeList[h];
}
```

Fig. 9. Algorithm to delete the TCAM entry in layer h.

```
// Insert prefix P = IP/len into the TCAM consisting of L layers.
Insert(IP, len, L)
{
01 result = LookupTCAM(IP, len, 0); // Search all layers
02 if (result.match = 0) $P_{match}$ = NULL;
03 else $P_{match}$ = IP/SRAM[result.idx].len;
04 if ($P_{match}$ = P) return; // P exists
05 if ($P_{match} \supset P$ or $P_{match}$ == NULL) {   The 2nd case described
06     Chain_InsertTCAM(IP, len, 1);                 in section 3.1.2. So, P
07     return;                                       is put in layer 1.
08 }
09 SearchedResult[1] = result;
10 lbnd = 2; ubnd = L;
11 while(lbnd <= ubnd) {
12     if (lbnd = ubnd) { // P is put in layer lbnd
13         Chain_InsertTCAM(IP, len, lbnd);
14         return;
15     }
16     mid = $\lceil (lbnd + ubnd)/2 \rceil$;
17     result = LookupTCAM(IP, len, mid);
18     SearchedResult[mid] = result;
19     if (result.match = 0) ubnd = mid – 1;
20     else {
21         $P_{match}$ = IP/SRAM[result.idx].len;
22         if ($P_{match}$ = P) return; // P exists
23         if ($P_{match} \supset P$) ubnd = mid – 1;
24         else lbnd = mid + 1; //$P \supset P_{match}$
25     }
26 }//end while
}
```

Fig. 10. Prefix insertion algorithm.

layer 1. In this case, no other TCAM entry is affected. If a match is found, the matched prefix P_m is denoted by $IP/SRAM$[*result.idx*].*len*. If $P_m = P$ or $P_m \supset P$ due to *len* $\geq$ *SRAM*[*result.idx*].*len*), P must be in layer 1 as stated in Section 3.1.2. Then, the function *Chain_InsertTCAM*(*IP*, *len*, 1) in Fig. 8 is called to insert P in layer 1.

After executing lines 1 to 9 of algorithm *Insert*(*IP*, *len*, *L*), the remaining task is to deal with the case in which $P \supset P_m$ due to *len* $<$ *SRAM* [*result.idx*].*len*. As described earlier, P_m must be in layer 1. Therefore, P may be inserted in layer 2 or higher. A binary search-like procedure is designed to determine the layer $k (k > 1)$ in which P should be inserted as shown in lines 10 to 26 of Fig. 10.

Initially, the low and high bounds of the layers to be checked are set as *lbnd* $= 2$ and *ubnd* $= L$. The while-loop starts to perform the function *Chain_InsertTCAM*(*IP*, *len*, *lbnd*) if *lbnd* $=$ *ubnd*. Then, the middle layer between *lbnd* and *ubnd* is searched as shown in lines 16 to 17. The result returned from a TCAM lookup is buffered in the array *SearchResult*[] to indicate that the layer *mid* has been searched and will be needed later. The following four possible cases have to be considered:

1. No match is found in layer *mid* (line 19): The search process must be continued by setting *ubnd* $=$ *mid* $-$ 1. P must not belong to layer *mid* $+$ 1 or higher because otherwise P should enclose a prefix in layer *mid* and contradict the condition that P is disjoint from all the prefixes in layer *mid*. P may possibly belong to layer *mid*. The search process in

```
// Delete prefix P = IP/len from TCAM containing L layers.
Delete(IP, len, L)
{
01  result = LookupTCAM(P, len, 0);
02  if (result.match == 0) return; // P does not exist;
03  else $P_m$ = IP/SRAM[result.idx].len; // The matched prefix $P_m$
04  if ($P_m$ = P) { DeleteLayer(IP, len, 1); return;}
05  if ($P_m \supset P$) return; // P does not exist;
06  lbnd = 2; ubnd = L;
07  while(lbnd <= ubnd) {
08        mid = $\lceil$(lbnd+ubnd)/2$\rceil$;
09        result = LookupTCAM(IP, len, mid);
10       SearchedResult [mid] = result,
11       if (result.match = 0){
12           if (lbnd = ubnd) return;
13           ubnd = mid – 1; // P belongs to layers lbnd to mid – 1;
14       } else {
15         $P_m$ = IP/SRAM[result.idx].len; // The matched prefix
16         if ($P_m$ = P) {DeleteLevel(IP, len, mid); return; }
17         if ($P_m \supset P$){
18             if (lbnd = ubnd) return;
19             ubnd=mid – 1; // P belongs to layers lbnd to mid–1;
20         } else { // $P \supset P_m$
21             if (lbnd = ubnd) return;
22             lbnd=mid + 1; // P belongs to layers mid–1 to ubnd;
23         }
24      }
25  }//end while
}
```

Fig. 11. Prefix deletion algorithm.

layers *lbnd* to *mid* $-$ 1 is used to find out if such is the case.

2. P is the same as the matched prefix P_{match} (line 22): No further action is required because P already exists.
3. P is enclosed by the matched prefix P_{match} (lines 23): As in case (1), the search process is continued by setting *ubnd* $=$ *mid* $-$ 1.
4. P encloses the matched prefix P_{match} (lines 24): P belongs to one of the layers *mid* $+$ 1 to *ubnd*. Thus *lbnd* is set to *mid* $+$ 1 and the search process continues.

3.4 Prefix Deletion

Fig. 11 shows the detailed algorithm *Delete*(*IP*, *len*, *L*) used to delete prefix $P = IP/len$ from the TCAM containing L layers. Similar to the prefix insertion algorithm, this algorithm also takes $O(\log L)$ steps to determine if P exists in the TCAM and in which layer P is located. The logic of finding the layer to which P belongs is the same as the prefix insertion algorithm. Therefore, the detailed description of the process is omitted. After P is found in layer h, the algorithm *DeleteLayer*(*IP*, *len*, *h*) in Fig. 12 is called to execute the deletion operations. Before P can be removed from layer h of the TCAM, we have to ensure that the prefix enclosure relationship in the layer grouping scheme is still maintained. First, we check if a prefix called *PP* exists in layer $h+1$ and if *PP* encloses P as shown in lines 2 to 5. If not, P can be deleted directly in line 7. Otherwise, we have to check if *PP* also encloses a prefix other than P in layer h. If so, P can be deleted directly in line 12 since all the prefixes still follow the prefix enclosure relationship in the layer grouping scheme. Otherwise, *PP* must be moved from layer $h+1$ to h because P is the only child of *PP*. The same process continues to delete *PP* in layer $h + 1$ after *PP* is moved to layer h in lines 14 to 16.

```
// Delete P=IP/len in layer h from TCAM containing L layers.
01  DeleteLayer(IP, len, h);
{
01  for (i = h; i < L; i++){
02     result = SearchedResult[i];
03     if (SearchedResult[i+1] = NULL)
04         result1 = LookupTCAM(IP, len, i+1);
05     else result1 = SearchedResult[i+1];
06      if (result1.match == 0) {
07          DeleteTCAM(result.idx, i); return;
08      } else { // PP in layer i+1 encloses P
09          TCAM[result.idx].V = 0; //temporarily delete P
10          resultx = LookupTCAM(IP, SRAM[result1.idx].len, i);
11          if (resultx.match == 1) {
12              DeleteTCAM(result.idx, i); return;
13          } else { //move TCAM[result1.idx] to layer i
14              TCAM_Write(result.idx, IP, len, i);
15              SRAM[result.idx].len = SRAM[result1.idx].len;
16               len = SRAM[result1.idx].len;}
17     }
18  }
}
```

Fig. 12. Delete a prefix in layer h.

4 Performance

In this section, the performance of the proposed TCAM update scheme and PLO_OPT and CAO_OPT proposed in [6] is evaluated. In the worst case, each update in CAO_OPT takes $L/2$ TCAM movements, which is better than the $L-1$ and W TCAM movements needed in the proposed scheme and PLO_OPT, respectively. The simulations are used to evaluate the average numbers of TCAM lookups and writes per insertion and deletion. Here, we show that the number of search cycles needed to complete the operations needed for each insertion and deletion is small and can be ignored.

We programmed all the schemes (the proposed scheme, PLO_OPT and CAO_OPT [6]) by using the C programming language. The five tables analyzed earlier are used in the performance evaluation. In our experiments, we divide the original routing table into two parts. One is used as the routing table, and the other is used as the insertion trace. The routing table is 95 percent of the prefixes in the original routing table and the insertion trace is the other 5 percent. First, we insert the routing table into TCAM, and the corresponding binary trie structure of CAO_OPT is constructed. A prefix is arbitrarily selected from the update trace as an insertion key. After all the insertion keys are inserted, we randomly select another 5 percent of the original prefixes as the deletion trace. We then randomly select one prefix at a time from the deletion trace to perform the deletion operations and obtain the performance results.

Tables 4 and 5 show the performance results for the insertions and deletions, respectively. The proposed

TABLE 4
Results for Insertion

Table	PLO_OPT		CAO_OPT		Proposed	
	# of lookups	# of writes	# of lookups	# of writes	# of lookups	# of writes
AS2.0	0	7.204	0	1.206	1.068	1.029
AS1221	0	5.426	0	1.189	1.076	1.039
AS6447	0	7.498	0	1.254	1.077	1.037
V6Gene1	0	10.837	0	1.407	1.084	1.079
V6Gene2	0	10.560	0	1.368	1.062	1.057

update scheme takes fewer writes than the PLO_OPT and CAO_OPT because the TCAM free space sits between layers 1 and 2. However, the write operations needed for PLO_OPT, CAO_OPT, or the proposed update algorithm cannot be performed in parallel with the TCAM search operations. In other words, when a write operation is executed, the TCAM search operations must be locked to prevent erroneous search results. As a result, PLO_OPT and CAO_OPT lose more lookup cycles than the proposed scheme due to the write operations. With the assumptions of using an invalid bit for each TCAM entry and the support of a dual ported memory and the mutual exclusion of a concurrent read and write, the TCAM search operations may not need to be locked when a TCAM write operation is ongoing.

Some additional lookups are required because the proposed update scheme does not use a local CPU and an off-chip memory to compute how to update the routing table. As shown in Tables 4 and 5, the average numbers of additional TCAM lookups per insertion and deletion are 1.062 to 1.084 and 2.135 to 2.523 respectively. As such, the proposed scheme takes only one to two additional lookups to compute how to update a prefix. On the other hand, the CAO_OPT scheme, which uses an extended binary trie to record the positions of the prefixes stored in TCAM, takes at least 300 CPU cycles to compute how to insert and delete prefixes based on our implementation of the CAO_OPT. If the local CPU runs at 1 GHz, 300 cycles amounts to 100 ns. These update operations on a local CPU can be executed in parallel with the TCAM lookup operations.

To quantify the overall performance, we introduce the following TCAM hardware assumptions.

1. The normal TCAM search operations as well as the ones for updates are executed on demand.

TABLE 5
Results for Deletion

Table	PLO_OPT		CAO_OPT		Proposed	
	# of lookups	# of writes	# of lookups	# of writes	# of lookups	# of writes
AS2.0	0	9.220	0	1.164	2.377	1.079
AS1221	0	7.558	0	1.144	2.255	1.064
AS6447	0	9.709	0	1.236	2.523	1.123
V6Gene1	0	11.915	0	1.371	2.254	1.064
V6Gene2	0	11.474	0	1.363	2.135	1.032

TABLE 6
Average Numbers of Cycles Needed to Perform 1,000 Insertion and 1,000 Deletion

Table	1000 insertions		1000 deletions		
	cycles for lookups	cycles for writes	cycles for lookups	cycles for writes	total
AS2.0	1,068	3,087	2,377	3,237	9,769
AS1221	1,076	3,117	2,255	3,192	9,640
AS6447	1,077	3,111	2,523	3,369	10,080
V6Gene1	1,084	3,237	2,254	3,192	9,767
V6Gene2	1,062	3,171	2,135	3,096	9,464

2. The TCAM memory width is 144 bits.
3. The TCAM has a sustained search rate of 360 million lookups per second [11].
4. A TCAM write takes three clock cycles [16].
5. A lookup takes one clock cycle.

We suppose that 1,000 insertions and 1,000 deletions occur per second and that current TCAM devices could provide a throughput of 360 million lookups per second. The average number of clock cycles of 1,000 insertions or deletions can be calculated as shown in Table 6, where each TCAM write is assumed to take three cycles. Based on the proposed TCAM coprocessor architecture, the update threads employ the cycles that the lookup threads generally use in traditional TCAM architecture. Only 9,464 to 10,080 TCAM search cycles are used to complete our update process, which amounts to only a small portion of the total TCAM search cycles (360 million cycles). By using these lookup cycles for updates, the local CPU and a large amount of the off-chip SRAM used to keep the auxiliary binary trie are no longer needed. As shown in Table 7, if the size of the IPv6 table is as large as 120 K prefixes, the auxiliary binary trie needs a memory of up to 35 MB. The memory needed for the proposed update scheme is much smaller than that of auxiliary binary trie. The proposed TCAM architecture becomes simpler than the traditional one. Therefore, its overall clock rate can be implemented faster.

5 Conclusions

In this paper, we described the problem of the TCAM updating algorithm and presented a novel TCAM management for IPv4/IPv6. Based on the analyzed prefix

TABLE 7
The Memory Required for the Auxiliary Binary Trie [6] and the Proposed Update Scheme

Table	Auxiliary binary trie			Proposed
	# of prefixes	# of nodes	Memory (KB)	Memory (KB)
AS2.0	3,001	19,604	402	3.27
AS1221	933	6,148	126	1.25
AS6447	3,090	21,778	446	3.36
V6Gene1	120,861	1,611,438	33,047	118.4
V6Gene2	127,697	1,707,504	35,017	125.1

enclosure relationship of the routing tables, the prefixes are divided into groups through a layered approach. Four unused bits in each TCAM entry are used for the *layer field* to facilitate the proposed update operations. As a result, no binary trie data structure, which is usually large, is needed. The performance results show that only a small portion of total TCAM search cycles is used to complete our update process. Overall, the proposed TCAM architecture is much simpler than traditional TCAM architecture.

Acknowledgments

The authors would like to express their sincere thanks to the editors and the reviewers, who gave very insightful and encouraging comments. This work was supported in part by the National Science Council, Republic of China, under Grant NSC-99-2221-E-006-105-MY3 and NSC-100-2221-E-006-141-MY3.

References

[1] J. Yu, V. Fuller, T. Li, and K. Varadhan, "RFC1519: Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy," *Internet Eng. Task Force (IETF)*, Sept. 1993.

[2] R. Hinden and S. Deering, "RFC3513: Internet Protocol Version 6 (IPv6) Dressing Architecture," *Internet Eng. Task Force (IETF)*, Apr. 2003.

[3] Z. Pfeffer, B. Gamache, and S.P. Khatri, "A Fast Ternary Cam Design for IP Networking Applications," *Proc. 12th Int'l Conf. Computer Comm. and Networks (ICCCN)*, 2003.

[4] K. Zheng, C.-C. Hu, H.-B. Liu, and B. Liu, "An Ultra High Throughput and Power Efficient TCAM Based IP Lookup Engine," *Proc. IEEE INFOCOM*, 2004.

[5] BGP Analysis Reports, http://bgp.potaroo.net/index-bgp.html, 2013.

[6] D. Shah and P. Gupta, "Fast Updating Algorithms for TCAMs," *IEEE Micro*, vol. 21, no. 1, pp. 36-47, Jan. 2001.

[7] K. Zheng and B. Liu, "A Scalable IPv6 Prefix Generator for Route Lookup Algorithm," *Proc. Int'l Conf. Advanced Information Networking Applications (AINA)*, 2006.

[8] M.A. Ranchez, E.W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Trans. Networking*, vol. 15, no. 2, pp. 8-23, Mar./Apr. 2001.

[9] M.J. Akhbarizadeh, M. Nourani, and C.D. Cantrell, "Prefix Segregation Scheme for a TCAM-Based IP Forwarding Engine," *IEEE Micro*, vol. 25, no. 4, pp. 48-63, July/Aug. 2005.

[10] Y.-M. Hsiao, M.-J. Chen, Y.-J. Hsiao, H.-K. Su, and Y.-S. Chu, "A Fast Update Scheme for TCAM-Based Ipv6 Routing Lookup Architecture," *Proc. the 15th Asia-Pacific Conf. Comm. (APCC '09)*, 2009.

[11] Renesas 20Mbit Standard TCAM R8A20410BG, http://www.renesas.com/media/products/memory/TCAM/p20_tcam_products.pdf, 2013.

[12] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey," *IEEE J. Solid-State Circuits*, vol. 41, no. 3, pp. 712-727, Mar. 2006.

[13] G. Huston, "Exploring Autonomous System Numbers," *Internet Protocol J.*, vol. 9, no. 1, Mar. 2006.

[14] G. Wang and N.-F. Tzeng, "TCAM-Based Forwarding Engine with Minimum Independent Prefix Set (MIPS) for Fast Updating," *Proc. IEEE Int'l Conf. Comm. (ICC '06)*, 2006.

[15] T. Mishra and S. Sahni, "DUOS—Simple Dual TCAM Architecture for Routing Tables with Incremental Update," *Proc. IEEE Symp. Computers and Comm. (ISCC '10)*, 2010.

[16] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," *Proc. ACM SIGCOMM*, pp. 193-204, 2005.

[17] M. Adiletta, M.R. Bluth, D. Bernstein, G. Wolrich, and H. Wilkinson, "The Next Generation of Intel IXP Network Processors," *Intel Technology J.*, vol. 6, no. 3, pp. 6-18, 2002.

[18] Z.J. Wang, H. Che, and S.K. Das, "CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking," *IEEE Trans. Computers*, vol. 53, no. 12, pp. 1602-1614, Dec. 2004.

[19] H. Song and J. Turner, "Fast Filter Updates for Packet Classification Using TCAM," *Proc. IEEE GLOBECOM*, 2006.

[20] T. Mishra, S. Sahni, and G.S. Seetharaman, "PC-DUOS: Fast TCAM Lookup and Update for Packet Classifiers," *Proc. IEEE Symp. Computers Comm. (ISCC)*, pp. 265-270, 2011.

[21] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," *Proc. IEEE INFOCOM*, Mar. 2003.

[22] Y.-K. Chang, "Power-Efficient TCAM Partitioning for IP Lookups with Incremental Updates," *Proc. Int'l Conf. Information Networking: Convergence in Broadband and Mobile Networking (ICOIN '05)*, pp. 531-540, Jan. 2005.

[23] Y.-K. Chang, C.-C. Su, Y.-C. Lin, and S.-Y. Hsieh, "Efficient Gray Code Based Range Encoding Schemes for Packet Classification in TCAM," *IEEE/ACM (ACM Trans. Networking*, vol. 21, no. 4, pp. 1201-1214, Aug. 2013.

Fang-Chen Kuo received the MS degree in computer science and information engineering from National Cheng Kung University, Taiwan, Republic of China, in 2006. He is currently working toward the PhD degree in computer science and information engineering at National Cheng Kung University, Taiwan, Republic of China. His current research interests include high-speed networks and high-performance Internet router design.

Yeim-Kuan Chang received PhD degree in computer science from Texas A&M University, College Station, in 1995. He is currently a professor in the Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan. His research interests include Internet router design, computer architecture, and multiprocessor systems.

Cheng-Chien Su received the MS and PhD degrees in computer science and information engineering from National Cheng Kung University, Taiwan, Republic of China, in 2005 and 2011, respectively. His research interests include high-speed packet processing in hardware and deep packet inspection architectures.