Chapter 1

Green TCAM-based Internet Routers

Tania Mishra

Department of CISE, University of Florida

Sartaj Sahni

Department of CISE, University of Florida

1.1 Introduction

Internet routers are devices that connect several packet switched networks to allow communication and resource sharing among a large group of users. A router implements the packet forwarding and routing functions of network layer which is the third layer of the seven-layer OSI model of computer networking. A router has several input and output ports through which it receives and sends packets. A packet arriving at an input port of a router is transfered to an appropriate output port and from here to its next hop on the path to its destination. A router maintains a list of rules in a forwarding table that is used to determine the next hops for packets during packet forwarding. A router updates its forwarding table continuously in response to changes in the Internet. All the routers in a network communicate with each other using routing protocols to remain updated of the changes and to select the best paths to reachable destination addresses.

As Internet packets get from source to destination they go thorough a number of routers. At each router, a forwarding engine uses the destination address of the packet and a set of rules to determine the next hop for the packet. A forwarding table often contains hundreds of thousands of rules. A packet forwarding rule (P, H) comprises a prefix P and a next hop H. A packet with destination address d is forwarded to H where H is the next hop associated with the rule that has the longest prefix matching d. Figure 1.1 shows a small forwarding table with 6 prefixes. The prefix associated with rule R5 is 100 (the * at the end indicates a sequence of don't care bits) and the associated next hop is H5. Rule R5 matches all destination addresses that begin with 100. The length of the prefix 100 associated with R5 is 3. A destination address that begins with 100 is matched by rules R1, R3, and R5. Among these rules, R5 is the one with the longest prefix. So, H5 is the next hop for packets with a destination address that begins with 100.

	Prefixes	Next Hop
R1	*	H1
R2	00*	H2
R3	10*	H3
R4	11*	H4
R5	100*	H5
R6	111*	H6

FIGURE 1.1: An example 6-prefix forwarding table

Figure 1.2 shows a block diagram of the internals of a router. The part of a router that deals with packet forwarding is called the data plane and the part that communicates with other routers and selects the best routes to different destinations and keeps the forwarding table updated is called the control plane. The control plane updates the forwarding table either in a batch or incrementally. When the updates are done in a batch, two copies of the forwarding table are maintained as shown in the Figure 1.3. While one copy is used for data plane lookups, the other copy is made up-to-date by the control plane. When the updating is complete, the data plane switches to the freshly updated copy for lookups, and the outdated copy is updated by the control plane. Thus batch updates require twice the memory for storing two copies of the forwarding table and introduce a latency between arrival of an update request and the time the update is incorporated. In contrast, when the updates are applied incrementally, lookups and updates are done on the same table as shown in Figure 1.4.

With the rapid growth of the Internet, the number of routers being used is increasing dramatically. While routers used at homes consume about 5-

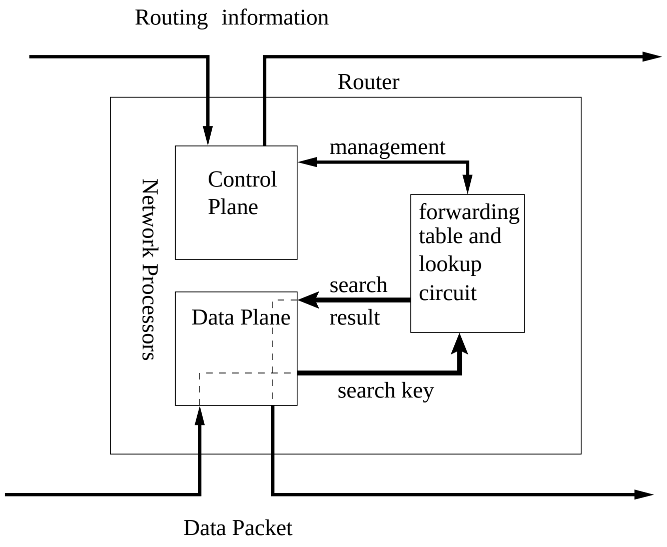


FIGURE 1.2: Control and data planes in routers

10W, edge routers consume about 4kW of power and the core routers about 10kW of power per rack [1]. It was measured that in Japan, ICT equipment in 2006 consumed about 45TWh or 4% of the electricity generated and 1% of the total energy consumption of the country [2]. Approximately, 25% of the ICT equipment energy is consumed by routers. The packet forwarding engine of a router needs a lot of energy to perform high-speed lookups and packet switching. For example, to keep up to a line rate of 40 Gbps, a packet forwarding engine must perform 125 million searches per second, assuming the minimum IPv4 packet size of 40 bytes. It was found that about 62% of the router power consumption happens in the packet forwarding engine.

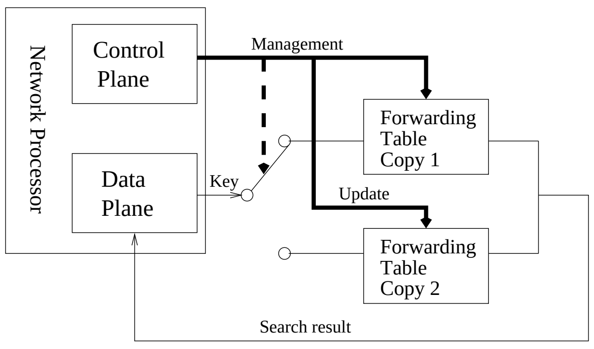


FIGURE 1.3: Router architecture with batch updating policy

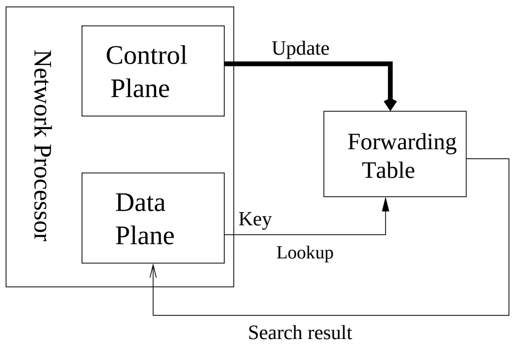


FIGURE 1.4: Router architecture with incremental updating policy.

With the growing trend of network usage, the total energy consumption in routers is only going to get worse. For example, with the current growth rate of Internet traffic at 40% per year in Japan, it is projected that by the year 2022, the energy consumption will exceed 10,000 TWh which was the total energy produced in Japan in 2005 [19]. Thus, unless we can make routers far more energy efficient, routers may soon consume most of the produced energy. This makes the design of low power packet forwarding engines essential.

In this chapter, we focus on reduction of energy consumption in a packet forwarding engine that uses a special type of memory called TCAM which is the acronym for Ternary Content Addressable Memory. A TCAM is different from a conventional memory in that, each bit may be set to one of the three states namely, 0, 1, and x (don't care). This makes it particularly convenient to store the destination prefixes along with their trailing sequence of 'x's. A TCAM has an associated SRAM which is used to store the next hops. A TCAM often has a priority encoder to choose the best match among multiple matches. TCAMs are attractive for use in edge and core routers, because like an associative memory, TCAMs enable a parallel search across all the rules and complete a table lookup in one clock cycle. Even though TCAMs support high-speed lookups, they are power hungry. So, a lot of research has been done to reduce the power consumption both in the hardware and software domains [7, 4, 3, 8, 6, 9, 10, 11, 12]. Pure hardware approaches for power reduction are presented in [9, 10, 11, 12]. The software approaches are characterized by removing redundancies in the prefixes of the forwarding table and compacting them using different techniques, so that the number of prefixes to be stored in the TCAM decreases [7, 13, 8]. Using an indexed TCAM organization, the authors in [3, 4, 6] could obtain significant savings in power consumption. This chapter presents some of these latter techniques.

1.2 Simple TCAM (STCAM)

This is the basic scheme in which a single TCAM with its associated SRAM and a priority encoder is used to store the forwarding table. The prefixes are stored in decreasing order of length and the corresponding next hops are stored in the associated SRAM. In case of multiple matches, the priority encoder chooses the first match (which can be done in one clock cycle), and uses the address of the matched entry to index into the SRAM and obtain the next hop. Thus, the entire lookup completes in one TCAM clock cycle returning the next hop corresponding to the longest matching prefix.

Figure 1.5 shows the prefix assignment to a STCAM for the forwarding table in Figure 1.1. The prefixes are stored in the TCAM in decreasing order of prefix length and the next hops are stored in corresponding words of an SRAM. Now suppose the router receives a packet whose destination address begins with 111. Assuming the TCAM and SRAM words are indexed beginning at 0, a TCAM lookup returns the TCAM index 1 for the longest matching prefix. This index is used to access the SRAM and H6 is returned as the next hop for the packet.

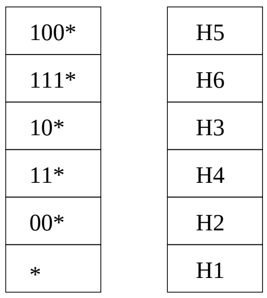


FIGURE 1.5: Rules of Figure 1.1 stored in a STCAM organization

1.3 Indexed TCAMs

In order to reduce power consumption during TCAM search, TCAM hardware supports a feature that searches only a portion of the TCAM. This reduces power consumption which is proportional to the size of the TCAM that needs to be searched. Zane et al. [3] proposed a two-level architecture in which the first level TCAM is called the index TCAM (ITCAM) and the second level TCAM is the data TCAM (DTCAM). The ITCAM is looked up using the destination address and the best match is used to activate a particular DTCAM segment of prefixes. The DTCAM segment is searched next for the longest matching prefix. To fill in the ITCAM and DTCAM segments with prefixes, Zane et al. [3] proposed two methods, namely subtree split and postorder split. Both these methods split a 1-bit trie representation of prefixes repeatedly till all the prefixes are entered into the DTCAM. In subtree split, variable sized DTCAM segments are created, with a maximum bounding size of b entries, where $b > 1$. The split is done in such a way that all the DTCAM segments, except one, contain between $\lceil b/2 \rceil$ and b entries. There is a single segment that contains between 1 and b prefixes. The 1-bit trie is traversed in a postorder fashion and while visiting a node v if it is found that the subtree rooted at v contains at least $\lceil b/2 \rceil$ prefixes and the subtree rooted at its parent (if any) contains more than b prefixes, then the subtree is splitted (or carved) at node v. The prefixes in the carved out subtree are inserted into a DTCAM segment in decreasing order of length. The prefix represented by the path from the root of the trie to node v is inserted in the ITCAM, and the corresponding ISRAM entry points to the start and end addresses of the DTCAM segment created for storing prefixes in subtree rooted at v. ITCAM prefixes are entered in the same order as they are generated upon post order traversal of the trie to fill up DTCAM segments. For a forwarding table with n prefixes, the number of ITCAM entries is at most $\lceil 2n/b \rceil$ and each bucket has at most $b+1$ prefixes (including the covering prefix[1]) of TCAM entries activated during a search, the power of each lookup is bounded as $P \leq 2n/b + b + 1$ which is minimized when $b = \sqrt{2n}$. The minimum power required is $2\sqrt{2n} + 1$ compared to n required for a STCAM. Lu and Sahni [4] observed that the subtree split algorithm is suboptimal producing, in the worst case, about twice the optimal number of DTCAM segments and correspondingly twice the optimal number of ITCAM entries. They improved upon the subtree splitting techniques of [3] by following a greedy postorder traversal policy.

The postorder split algorithm of Zane et al. [3] generates buckets with a fixed number of prefixes. Each bucket additionally includes up to W covering prefixes, where W is the length of the longest prefix in the forwarding table. All buckets, except one, contains exactly b forwarding table prefixes, whereas the remaining bucket has fewer than b forwarding-table prefixes. This remaining bucket can be padded with empty TCAM entries to make it the same size as b. All the buckets could contain a variable number of covering prefixes up to a maximum of W covering prefixes. The algorithm for post order split is different than subtree split in that now prefixes from several subtrees can be used to fill up a DTCAM segment of size b. As a result there may be more prefixes in the ITCAM, where multiple ITCAM prefixes point to the same DTCAM bucket. Note also that a bucket may contain up to 1 covering prefix for each subtree packed into it. Figure 1.6 shows the ITCAM, ISRAM,

[1]The covering prefix for v is the longest-length forwarding table prefix that matches all destination addresses of the form $P*$, where P is the prefix defined by the path from the root of the 1-bit trie to v.

DTCAM, and DSRAM configurations for the 6-prefix example of Figure 1.1. Lu and Sahni in [4] have proposed a greedy heuristic for creating fixed size buckets which improves upon the number of prefixes to be entered into the ITCAM. For this greedy heuristic, the worst case number of ITCAM prefixes is $\lceil \log_2(b) \rceil \times \lceil n/b \rceil$.

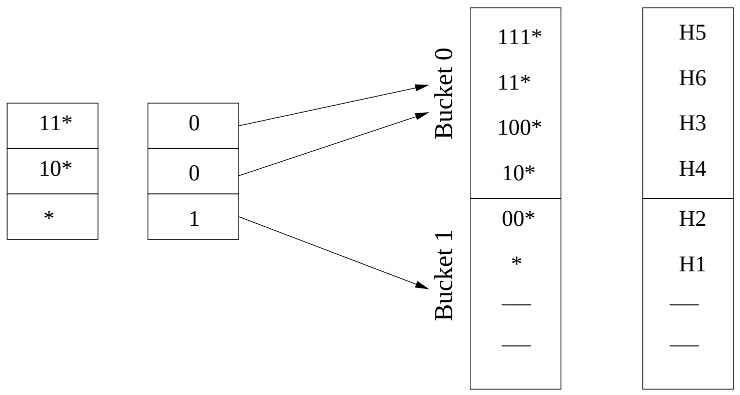


FIGURE 1.6: Six rules of Figure 1.1 stored in an indexed TCAM organization

1.4 TCAMs with Wide SRAMs

Lu and Sahni in [4] proposed the use of wide SRAMs in association with a TCAM. It was observed that the number of next hops for a router is usually quite small (same as the number of output ports), and so it requires a few bits to encode these next hops. In that case, the remaining space in an SRAM word remains un-utilized. Moreover, if a QDRII SRAM is used in association with a TCAM, then, 72 bits of SRAM memory could be accessed (in case of dual burst) or even 144 bits (quad burst) could be accessed simultaneously. To better utilize the SRAM space, the authors proposed to store a subtree of the 1-bit trie of prefixes in a forwarding table, which in turn reduces the number of TCAM entries and hence the power required for a lookup. Even though there is a minor increase in the lookup time due to additional processing needed to retrieve the nexthop from a wide SRAM word, it is still possible to complete a lookup in one clock cycle [8]. To store a subtree in an SRAM word, Lu and Sahni proposed a suffix node format which was later updated by Mishra and Sahni in [8] and is included in Figure 1.7. A suffix node must fit in an SRAM word.

To fill up a suffix node, a subtree of the 1-bit trie T is carved and the

Match start position	Suffix count	next hop of S0	len (S1)	S2	next hop of S1	...	len (Sk)	Sk	next hop of Sk	unused

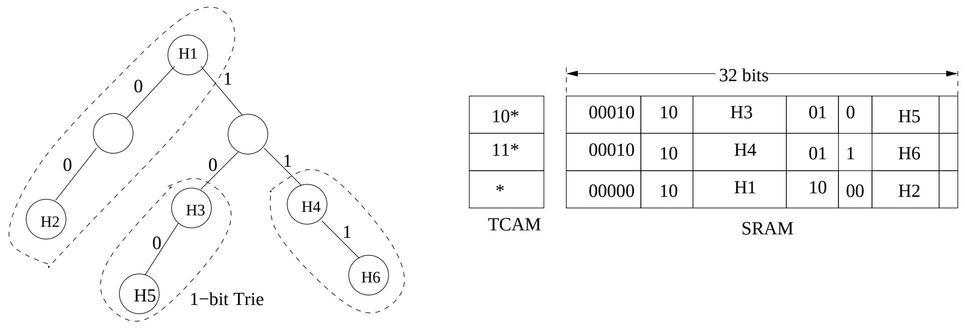
FIGURE 1.7: Suffix node of [4] with a 5-bit match start position field and an optimized representation of the first suffix.

size of the subtree is determined by the size of an SRAM word. If N is the root of the subtree being carved then $Q(N)$ is the prefix defined by the path from the root of the trie to the node N. Prefix $Q(N)$ is entered in the TCAM. Let $P_1, \cdots P_k$ be the prefixes in the subtree rooted at N, excluding the prefix stored at N, if any. Then the data collected to create a suffix node are:

1. Length $|Q(N)|$ of the prefix stored in TCAM.
2. Suffix count $k{+}1$, due to prefixes $P_1, \cdots P_k$ and either the prefix stored at node N or the covering prefix of node N.
3. Default next hop, which is the next hop corresponding to prefix (if any) stored at node N. Otherwise, the default next hop is set to the next hop of the covering prefix for node N. This field is marked as "next hop of S_0" in Figure 1.7.
4. For each prefix P_i, the suffix S_i of P_i consisting of bits from position $|Q(N)| + 1$ to $|P_i|$.
5. For each prefix P_i, the length of the suffix S_i, $|Si| = |Pi| - |Q(N)|$.
6. For each prefix P_i, its next hop.

Let u be the number of bits allocated to the prefix length, suffix count and default next hop fields (items 1, 2 and 3 of the above list) of a suffix node and let v be the sum of the number of bits allocated to a suffix length and next hop fields (items 5 and 6 of the above list). Let $len(S_i)$ be the length of the suffix S_i. The space needed by the suffix node fields for $S_1 \cdots S_k$ is $u + kv + \sum_{i=1}^{k} len(Si)$ bits. Thus it is required that $u + kv + \sum_{i=1}^{k} len(Si)$ be less than or equal to the bandwidth (or word size) of the SRAM.

Example 1 *Consider the 6-prefix forwarding table of Figure 1.1. Suppose that a suffix node is 32 bits long and the SRAM word size is also 32 bits. Suppose 5 bits are used for length of prefix matched which allows a prefix in TCAM to be 31 bits in length, 2 bits for the suffix count field (this allows up to 4 suffixes in a node as the count must be more than 0), 2 bits for the suffix length field (permitting suffixes of length up to 3), and 10 bits for a next hop (permitting up to 1024 different next hops). With this bit allocation, a suffix node may store up to 2 next hops, including the default next hop. Figure 1.8 shows an example of a carving of the 1-bit trie for 6-prefix example and the TCAM and SRAM bit assignments. This carving has the property that no subtree needs a* *covering prefix and each subtree may be stored in a suffix node using the stated format.*



TCAM
10*
11*
*

SRAM						
00010	10	H3	01	0	H5	
00010	10	H4	01	1	H6	
00000	10	H1	10	00	H2	

FIGURE 1.8: Suffix node example

To search for a prefix, the TCAM is first looked up for the longest $Q(N)$ matching destination address d. Next, the SRAM word corresponding to the TCAM match is searched for the longest matching suffix. While matching the suffixes to d, the first $|Q(N)|$ bits of d are ignored and the matching is done starting from the $|Q(N)|+1$ bit of d. The nexthop corresponding to the longest matching suffix is returned from the SRAM word search. In case there is no suffix that matches d, the default next hop is returned from the SRAM word.

If the average number of prefixes packed into a suffix node is a_1, then the TCAM size is approximately n/a_1, where n is the total number of forwarding table prefixes. So, the power needed for a lookup in a forwarding table using a TCAM with a wide SRAM is about $1/a_1$ that required when the STCAM organization of Figure 1.1 is used.

For the carving heuristic, suppose u and v are as above and let w be the size of a suffix node. Also, for any node x in the 1-bit trie, let $ST(x)$ be the subtree rooted at x. Let $ST(x).numP$ be the number of prefixes in $ST(x)$ and let $ST(x).numB$ be the number of bits needed to store the suffix lengths, suffixes and next hops for these prefixes of $ST(x)$. For each node x the quantities $ST(x).numP$ and $ST(x).numB$ are computed by a postorder traversal of the trie. When x is null, $ST(x).numP = ST(x).numB = 0$. When x is not null, let l and r be its two children (either or both may be null). The following recurrence is obtained for $ST(x).numB$.

$$ST(x).numB = ST(l).numB + ST(l).numP + ST(r).numB + ST(r).numP \tag{1.1}$$

Since each prefix in $ST(l)$ and $ST(r)$ has a suffix that is 1 bit longer, the extra suffix bits are accounted for in x by adding $ST(l).numP+ST(r).numP$. In addition to the extra suffix bits, x needs to also store the suffix length and nexthop fields which carry over from its left and right children. So, in total

```
Algorithm visit(x)
{
    if (ST(x).size < w) return;
    if (ST(x).size == w) {split(x); return;}
    // ST(x).size > w
    if (x has only 1 non-null child y) {split(y); return;}
    // x has 2 non-null children y and z
    if (ST(y).numB ≤ ST(z).numB) {
        split(z);
        recompute ST(x).size;
        if (ST(x).size < w) return;
        if (ST(x).size == w) split(x);
        else split(y);
    }
    else // ST(y).numB > ST(z).numB
    // this is symmetric to the case ST(y).numB ≤ ST(z).numB
}
```

FIGURE 1.9: Visit function for subtree carving heuristic

$ST(l).numB + ST(l).numP + ST(r).numB + ST(r).numP$ bits are needed to store the suffix length fields, suffixes and nexthops.

The size, $ST(x).size$, of the suffix node needed by $ST(x)$ is given by

$$ST(x).size = ST(x).numB + u \tag{1.2}$$

The correctness of Equation 1.2 follows from the observation that in either case, u additional bits are needed for the prefix length, suffix count and the default next hop.

Example 2 *Lets take the trie for the 6-prefix forwrading table in Figure 1.1 and compute $ST(x).numP$, $ST(x).numB$ and $ST(x).size$. To keep things simple, for $ST(x).numB$ we consider the bits to store nexthop and suffix, ignoring the bits for the suffix length field. Similarly, for u in Equation (2), we count only the number of bits required to store the default nexthop. Figure 1.10 shows the triplets ($ST(x).numP$, $ST(x).numB$, $ST(x).size$) for each node. For example, if the node at 00 storing nexthop H2 is carved, then the corresponding SRAM word will just store the default nexthop H2. Hence, $ST(x).size$ = 10, assuming we use 10 bits to store a next hop. Similarly, if the node at 1 is carved, the corresponding SRAM word will store four suffixes, which are: 0, 1, 00, 11. These suffix bits, along with the their nexthops, will occupy 46 bits of SRAM space. In this case, the default hop $H1$ comes from the covering prefix and $ST(x).size$ = 46+10 = 56.*

The carving heuristic performs a postorder traversal of the 1-bit trie T using the visit algorithm of Figure 1.9. Whenever a subtree is split from the

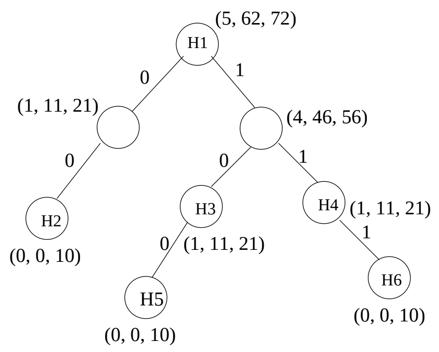


FIGURE 1.10: ($ST(x).numP$, $ST(x).numB$, $ST(x).size$) computed for each node

1-bit trie, the prefixes in that subtree are put into a suffix node and a TCAM entry for this suffix node generated. The complexity of the visit algorithm (including the time to recompute $ST(x).size$) is $O(1)$. So, the overall complexity of the tree carving heuristic is $O(nW)$, where n is the number of prefixes in the forwarding table and W is the length of the longest prefix.

It is possible to obtain further reduction in TCAM power by using the indexed TCAM schemes of Section 1.3 on TCAMs with wide SRAMs. The details of the method of adding an index TCAM to a TCAM with wide SRAM, can be found in [4]. Due to the necessity for storing covering prefixes in an indexed TCAM as well as in a wide SRAM, incremental update algorithms tend to be complex. For example, when a new prefix is added, it must be checked if the new prefix should replace any existing covering prefix. Similarly when a prefix is deleted from the forwarding table, it may have to be deleted from multiple locations, triggering fresh computations for covering prefixes. On the other hand, batch updating algorithms work perfectly for indexed TCAMs as well as those with wide SRAMs.

1.5 DUO

DUO is a dual TCAM architecture for faster prefix lookup and efficient storage. DUO uses advanced memory management schemes for inserting and deleting prefixes to and from the TCAM. The different versions of DUO are: 1) DUOS which is dual TCAM with simple SRAM, where both the TCAMs have a simple associated SRAM used for storing next hops. 2) DUOW which is dual TCAM with wide SRAM, where one or both the TCAMs have wide associated SRAMs that are used to store suffixes as well as next hops. 3) IDUOW which is indexed dual TCAM with wide SRAM, where either or both TCAMs have an associated index TCAM. All DUO versions support efficient incremental updates that do not degrade lookup speed.

1.5.1 DUOS

DUOS is a simple dual TCAM architecture, in which two TCAMs with associated SRAMs are used to store the prefixes in a forwarding table, as shown in the Figure 1.11. The TCAMs are labeled as ITCAM (Interior TCAM) and LTCAM (Leaf TCAM). A prefix in a forwarding table rule is stored in either ITCAM or LTCAM. DUOS uses a reasonably efficient data structure, such as a binary trie, to store the prefixes in the control plane. This is needed for the initial prefix assignment to the TCAMs as well as for prefix assignment during incremental updates. A quick evaluation of a binary trie stored in a 100ns DRAM shows that it permits about 300K IPv4 lookups, inserts, and deletes per second. This performance is adequate for the anticipated tens of thousands of control plane operations. The prefixes stored in the leaf nodes of the trie are entered in the LTCAM, and the remaining prefixes are entered in the ITCAM. Since the LTCAM stores prefixes found in the leaf nodes of the trie, the prefixes in the LTCAM are disjoint and so at most one may match any given destination address. Consequently, the LTCAM prefixes, even though of varying length, may be stored in any order. Further, the LTCAM does not require a priority encoder and, as a result, the latency of an LTCAM search is up to 50% less than that of a search in a TCAM with a priority encoder [5]. A data plane lookup is performed by doing a search for the packet's destination address in both ITCAM and LTCAM. The ITCAM search yields the next hop associated with the longest matching non-leaf prefix while the LTCAM search yields the next hop associated with at most one leaf prefix that matches the destination address. Additional logic shown in Figure 1.11 returns the next hop (if any) from the LTCAM search; the next hop from the ITCAM search is returned only if the LTCAM search found no match. Note that since the LTCAM has no priority encoder, its search completes sooner than that in the ITCAM. The combining logic of Figure 1.11 can take advantage of this observation and abort the ITCAM search whenever the LTCAM search is successful, thereby reducing average lookup time. The correctness of the lookup is readily established. It was found from a number of forwarding tables downloaded from [18] and [17] that over 90% of the prefixes are stored in the LTCAM and less than 10% are stored in the ITCAM. Consequently, the LTCAM services most of the lookup requests.

Figure 1.12 shows the binary trie stored in the control plane for our 6-prefix forwarding table of Figure 1.1 together with the content of the two TCAMs and the two SRAMs of DUOS. Each node of the control plane trie has fields such as *prefix*, *slot*, *nexthop* and *length* which aid in storing the prefix (if any) on that node, along with the ITCAM or LTCAM slot in which the prefix is stored and the nexthop and length of the prefix. Functions for basic operations

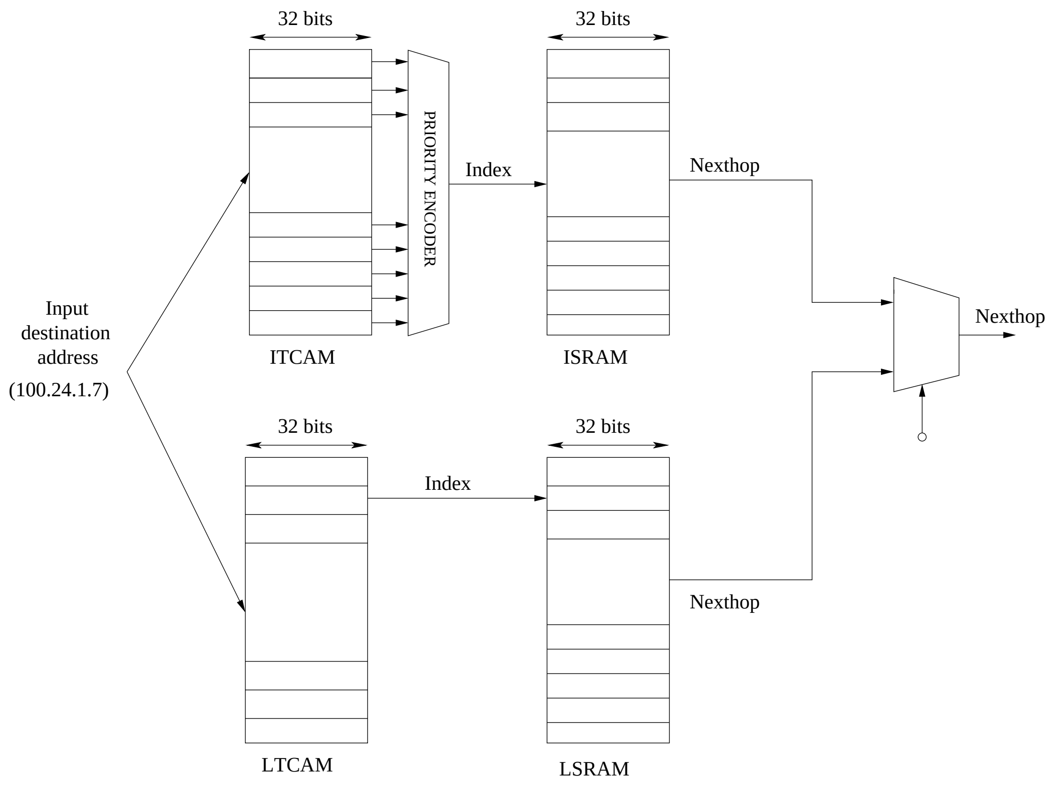


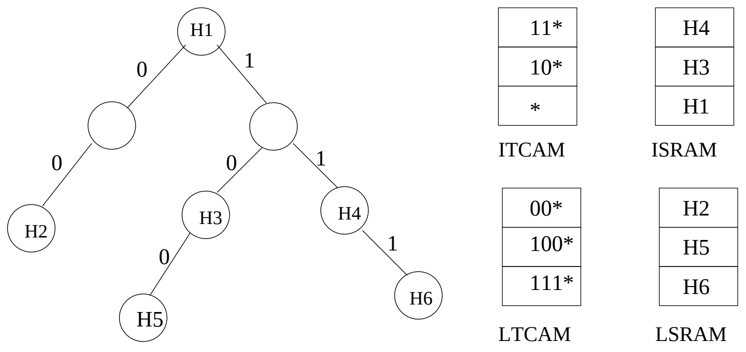
FIGURE 1.11: Dual TCAM with simple SRAM

on the control plane trie (hereinafter simply referred to as trie) are assumed (see Figure 1.13).

As the control plane will modify the ITCAM, LTCAM, ISRAM, and LSRAM while the data plane performs lookups, the TCAMs need to be dual ported. Specifically, the following assumptions are made:

1. Each TCAM has two ports, which can be used to simultaneously access the TCAM from the control plane and the data plane.
2. Each TCAM entry/slot is tagged with a valid bit, that is set to 1 if the content for the entry is valid, and to 0 otherwise. A TCAM lookup engages only those slots whose valid bit is 1. The TCAM slots engaged in a lookup are determined at the start of a lookup to be those slots whose valid bits are 1 at that time. Changing a valid bit from 1 to 0 during a data plane lookup does not disengage that slot from the ongoing lookup. Similarly, changing a valid bit from 0 to 1 during a data plane lookup does not engage that slot until the next lookup.

The availability of the function $waitWriteValidate$ which writes to a



(a) Trie (b) DUOS

FIGURE 1.12: DUOS for the 6-prefix forwarding table of Figure 1.1. Note that prefixes in ITCAM are stored in length order, whereas those in LTCAM are stored arbitrarily.

Function: Trie.insert
(a, b) = Trie.insert(prefix, length, nextHop);
This function inserts a prefix given its length and next hop into the control-plane binary trie. It returns the trie node a which stores the new prefix and a's nearest ancestor node b that contains a prefix.
Function: Trie.delete
(a, b) = Trie.delete(prefix, length);
This function deletes a prefix from the control plane trie and returns the trie node a that used to store the prefix just deleted and a's nearest ancestor node b that contains a prefix.
Function: Trie.change
a = Trie.change(prefix, length, newHop);
This function changes the next hop associated with a prefix and returns the trie node a that contains the prefix.

FIGURE 1.13: Table of control-plane trie functions

TCAM slot and sets the valid bit to 1, is assumed. In case the TCAM slot being written to is the subject of ongoing data plane lookup, the write is delayed till this lookup completes. During the write, the TCAM slot being written to is excluded from data plane lookups[2]. This is equivalent to the requirement that "After a rule is matched, resetting the valid bit has no effect

[2]A possible mechanism to accomplish this exclusion is to set the valid bit to 0 before commencing the write and to change this bit to 1 when the write completes.

on the action return process" [15], and to setting the valid entry to "hit" [14]. Similarly, the availability of the function $invalidateWaitWrite$ is assumed, which sets the valid bit of a TCAM slot to 0 and then writes an address to the associated SRAM word in such a way that the outcome of the ongoing lookup is unaffected.

Note that $waitWriteValidate$ may, at times, write the prefix and nexthop information in the TCAM and associated SRAM slot and validate it, without any wait. This happens, for example, when the writing is to be done to a TCAM slot that is not the subject of the ongoing data plane lookup. The wait component of the function $waitWriteValidate$ is said to be null in this case.

Figure 1.14 lists the various update algorithms we define later in this section for DUOS and its associated ITCAM and LTCAM. The indentation represents the hierarchy of function calls. A function at one level of indentation calls one or more functions below it at the next level of indentation or at the same level of indentation.

```
DUOS:
insert
delete
change
    ITCAM (with simple SRAM):
    insert
    delete
    change
        getSlot
        freeSlot
            movesFromAbove
            movesFromBelow
            getFromAbove
            getFromBelow
    LTCAM (with simple SRAM)
    insert
    delete
    change
```

FIGURE 1.14: Table of functions used for incremental update

1.5.2 DUOS Incremental Update Algorithms

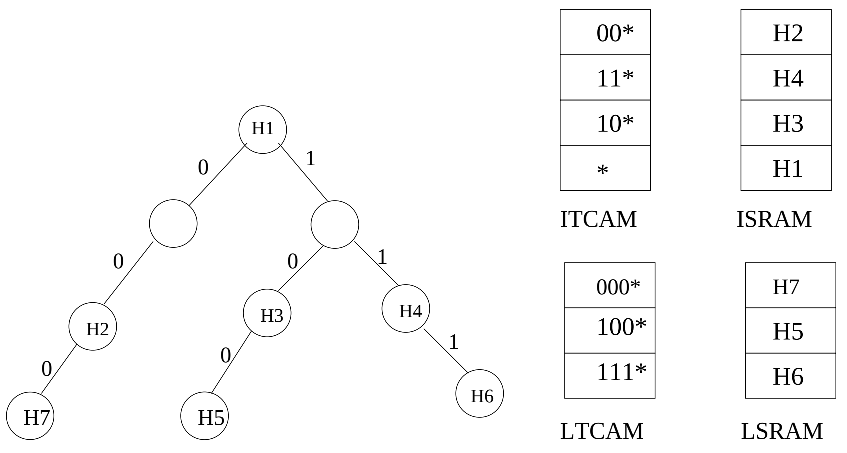
1.5.2.1 Insert

Figure 1.15 gives the algorithm to insert a new prefix p of length l and nexthop h. For simplicity, it is assumed that p is, in fact new (i.e., p is not already in the rule table). First, p is inserted into the trie using the trie

```
Algorithm: insert (p, l, h)
 (m, n) = Trie.insert(p, l, h)
 if m is a leaf then begin
     if n exists and n→prefix was a leaf prefix then
         slot = ITCAM.insert(n→prefix, n→nexthop, n→length);
                     // n→prefix is no longer a leaf
         LTCAM.delete(n→slot);
         n→slot = slot;
     endif
     m→slot = LTCAM.insert(p, h, l);
 else m→slot = ITCAM.insert(p, h, l);
 endif
```

FIGURE 1.15: Algorithm to insert into DUOS

insertion algorithm, which returns nodes m and n, where m is the trie node storing p and n is the nearest ancestor (if any) of m that has a prefix. When m is a leaf of the trie, there is a possibility that the insertion of p transformed a prefix that was previously a leaf prefix into a non-leaf prefix. If so, this prefix is moved from the LTCAM to the ITCAM. Regardless, p is inserted into the LTCAM. When m is not a leaf, p is inserted into the ITCAM. Figure 1.16 illustrates the insertion of a new rule (000*, H7) to the 6-prefix forwarding table of Figure 1.1.



(a) updated trie (b) updated DUOS

FIGURE 1.16: Insert new rule {000*, H7} to the 6-prefix table.

```
Algorithm: delete ($p$, $l$)
  ($m$, $n$) = Trie.delete($p$, $l$)
  If $m$ is a leaf then
    LTCAM.delete($m\rightarrow slot$)
    If $n$ exists and $n$ is now a leaf then
      $slot$ = LTCAM.insert($n\rightarrow prefix$, $n\rightarrow nexthop$, $n\rightarrow length$)
      ITCAM.delete($n\rightarrow slot$, $n\rightarrow length$) // since $n$ is now a leaf prefix
      $n\rightarrow slot = slot$;
    endif
  else
    ITCAM.delete($m\rightarrow slot$, $m\rightarrow length$)
  endif
```

FIGURE 1.17: Algorithm to delete from DUOS

1.5.2.2 Delete

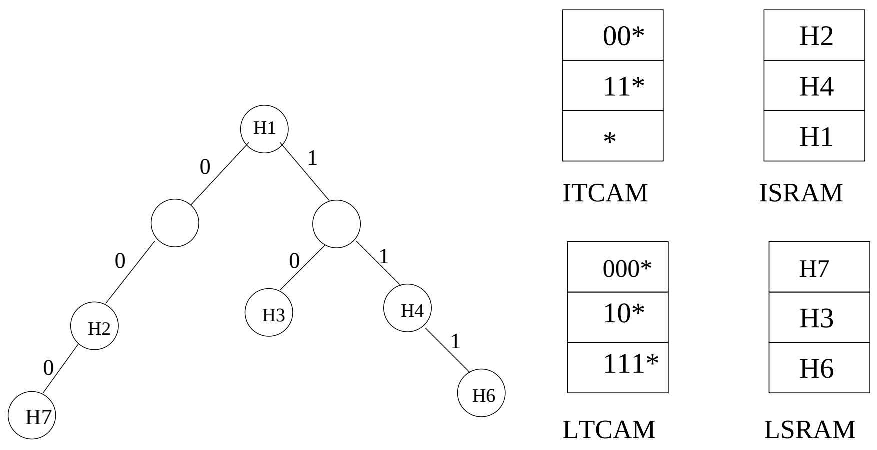
Figure 1.17 gives the algorithm to delete the prefix p from DUOS. For simplicity, it is assumed that p is, in fact, present in the rule table and so may be deleted. First, p is deleted from the trie. The trie deletion function returns nodes m and n, where m is the trie node where p was stored and n is the nearest ancestor (if any) of m that has a prefix. If m was a leaf, then p is to be deleted from the LTCAM. In this case, the prefix (if any) in n may become a leaf prefix. If so, the prefix in n is to be moved from the ITCAM to the LTCAM. When m is not a leaf, p is deleted from the ITCAM. Figure 1.18 illustrates the delete procedure for rule {100*, H5} from the 7-prefix forwarding table of Figure 1.16.

1.5.2.3 Change

To change the nexthop of an existing prefix to $newH$, first, the next hop of the prefix in the trie is changed and the node m that contains p is returned. Then, depending on whether m is a leaf or non leaf, the change function is invoked for the corresponding TCAM. Figure 1.19 gives the algorithm.

1.5.3 ITCAM Algorithms

The prefixes in the ITCAM are stored in a manner that supports the determination of the longest matching prefix (i.e., in any topological order that conforms to the precedence constraints defined by the binary trie–$p1$ must come before $p2$ whenever $p1$ is a descendent of $p2$ [16]). Decreasing order of length is a commonly used ordering. The function $getSlot(length)$ returns an ITCAM slot such that insertion of the new prefix into this slot satisfies the ordering constraint in use provided the new prefix has the specified length; the function $freeSlot(slot, length)$ frees a slot previously occupied by a prefix of the specified length and makes this slot available for reuse later. These



(a) updated trie (b) updated DUOS

FIGURE 1.18: Delete rule {100*, H5} from the prefixes in Figure 1.16

```
**Algorithm: change** ($p$, *length*, *newH*)
    $m$ = Trie.change($p$, $l$, $newH$)
    If $m$ is a leaf then
        $m \to slot$ = LTCAM.change($p$, $m \to slot$, $newH$);
    else
        $m \to slot$ = ITCAM.change($p$, $m \to slot$, $newH$, $length$);
```

FIGURE 1.19: Algorithm to change a next hop in DUOS

```
Algorithm: insert(prefix, nexthop, length)
    slot = getSlot(length);
    ITCAM.$waitWriteValidate$(slot, prefix, nexthop);
    return slot;

Algorithm: delete(slot, length)
    freeSlot(slot, length);

Algorithm: change(prefix, oldSlot, nexthop, length)
    slot = insert(prefix, nexthop, length);
    delete(oldSlot, length);
    return slot;
```

FIGURE 1.20: ITCAM algorithms

functions, which are described in Section 1.5.5, are used in our ITCAM insert, delete, and change algorithms presented in Figure 1.20.

Notice that following the first step of the change algorithm, the prefix whose next hop is being changed is in two valid slots of the ITCAM–*oldSlot* and *slot*. This duplication does not affect correctness of data plane lookups as whichever one is matched by the ITCAM, the returned next hop that is valid either before or after the change operation. On the other hand, if an attempt was made to change the next hop in $ISRAM[oldSlot]$ directly, an ongoing lookup may return a garbled next hop. Similarly, if an entry is first deleted and then inserted, lookups that take place between the delete and the insert may return a next hop that doesn't correspond to the forwarding table state either before or after the change. If a $waitWriteValidate$ is used to change $ISRAM[oldSlot]$ to nexthop, *oldSlot* becomes unavailable for data plane lookups during the write operation and inconsistent results are returned in case the prefix in TCAM[*oldSlot*] is the longest matching prefix.

1.5.4 LTCAM Algorithms

The prefixes in the LTCAM are disjoint and so may be stored in any order. The unused (or free) slots of the LTCAM/LSRAM are linked together into a chain using the words of the LSRAM to build this chain. AV is used to store the index of the first LSRAM word on the chain. So, the free slots are AV, $LSRAM[AV]$, $LSRAM[LSRAM[AV]]$, and so on. The last free slot on the AV chain has $LSRAM[last] = -1$. The LTCAM algorithms to insert, delete, and change are given in Figure 1.22. These algorithms are self explanatory.

Example 3 *Figure 1.21 shows a small LTCAM with 5 entries. Two out of the five entries store prefixes, whereas the other three are empty. So, AV, which is stored at a memory location of the control plane, is set to 1. LSRAM[1] is set to 3, and LSRAM[3] is set to 4. Since the word at the address 4 is the last*

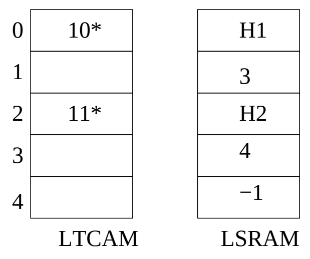


FIGURE 1.21: LTCAM-LSRAM layout showing free space management

```
Algorithm: insert(prefix, nexthop, length)
    if ($AV == -1$) throw NoSlotException;
    slot = AV;
    AV = LSRAM[slot];
    LTCAM.$waitWriteValidate$(slot, prefix, nexthop);
    return slot;

Algorithm: delete(slot)
    LTCAM.$invalidateWaitWrite$(slot, AV); // AV is stored in LSRAM[slot]
                                   // after waiting for an ongoing lookup to complete
    AV = slot;

Algorithm: change(prefix, oldSlot, nexthop, length)
    slot = insert(prefix, nexthop, length);
    delete (oldSlot);
    return slot;
```

FIGURE 1.22: LTCAM algorithms

free word, LSRAM[4] is set to -1. Now, as prefixes are inserted and deleted, AV and the values stored in the free LSRAM words are changed accordingly.

1.5.5 ITCAM Memory Management

The memory management scheme DLFS_PLO (Distributed and Linked Free Space with Prefix Length Ordering Constraint) offers the best algorithm known to us in terms of the number of prefix moves required when a prefix is to be inserted into or deleted from a TCAM [6]. Note that prefix moves are required since the ITCAM stores prefixes that could either enclose or be enclosed by other prefixes. Hence, to ensure that the first prefix matched in the TCAM is also the longest matching prefix, prefix insertions and deletions must be done carefully.

The description of memory management scheme DLFS_PLO includes an implementation of the *getSlot* and *freeSlot* functions used in Section 1.5.3

```
Algorithm: move (src, dest)
    ITCAM.waitWriteValidate(dest, ITCAM[src], ISRAM[src]);
```

FIGURE 1.23: Move from ITCAM[*src*] to ITCAM[*dest*]

```
Algorithm: getSlot(len)
    aP=0; bP=0; aC=0; bC=0;
    if (AV[len] == -1)
    // AV[len] stores the first free space in block of length len
        ma = movesFromAbove(len, &aP, &aC);
        mb = movesFromBelow(len, &bP, &bC);
        if (ma < mb)
            d = getFromAbove(len, aP, aC);
            if (top[len] > bot[len]) bot[len] = d;
            top[len] = d;
        else
            if (mb == $W + 1$) throw NoSpaceException; // no space
            d = getFromBelow(len, bP, bC);
            if (top[len] > bot[len]) top[len] = d;
            bot[len] = d;
        endif
    else
        d = AV[len];
        AV[len] = next[d];
    endif
    return d;
```

FIGURE 1.24: DLFS_PLO algorithm to get a free slot to insert a prefix whose length is *len*

to get and free ITCAM slots. The implementations employ the function *move* (Figure 1.23) that moves the content of an in-use ITCAM slot to a free ITCAM slot in such a way as to maintain data plane lookup consistency. DLFS_PLO maintains the invariant that an ITCAM slot has its valid bit set to 0 iff that slot wasn't matched by the ongoing data plane lookup (if any); that is, iff the slot isn't involved in the ongoing data plane lookup.

Lookup consistent implementations of *getSlot* and $freeSlot$ employ the following variables. W is the maximum prefix length (32 for IPv4), *top*[i] is the slot where the first prefix of length i is stored and *bot*[i] is the slot where the last prefix of length i is stored, $0 \le i \le W$ (i.e., these variables define the start and end of block i). Note that *top*[i] $\le$ *bot*[i] for a non-empty block i and *top*[i] $>$ *bot*[i] for an empty block. Let, *top*[0] $=$ *bot*[0] $= N + 1$ and *top*[$W + 1$] $=$ *bot*[$W + 1$] $= -1$. For an empty ITCAM, *top*[i] $= N + 1$ for $1 \le i \le W$; *bot*[i] $= -1$ for $1 \le i \le W$. The forward links, called *next*[], of the doubly-linked list are maintained using the ISRAM words corresponding to

```
Algorithm: freeSlot(d, len)
    if (top[len] == d) ITCAM[top[len]++].valid = 0;
    else if (bot[len] == d) ITCAM[bot[len]−−].valid = 0;
    else
        ITCAM.invalidateWaitWrite(d, AV[len]);
                // AV[len] is stored in ISRAM[d].
        if (AV[len] != -1) prev[AV[len]] = d;
        AV[len] = d;
    endif
```

FIGURE 1.25: DLFS_PLO algorithm to free a slot

the free ITCAM slots with $AV[i]$ recording the first slot on the list for the ith block. The backward links, called *prev*[], are maintained in these ISRAM words in case an ISRAM word is large enough to accommodate two links and in the control plane memory otherwise. All variables, including the array $AV[]$, are stored in the control plane memory.

The *getSlot* algorithm (Figure 1.24) first attempts to make available a slot from the doubly-linked list for the desired block. When this list is empty, the algorithm provides a free slot from either block boundary when there is a free slot on the block boundary. Otherwise, it moves a free slot from the nearest block boundary that has a free slot. This algorithm utilizes several supporting algorithms that are given in Figures 1.26 and 1.27. The algorithm *movesFromAbove* (*movesFromBelow*) returns the number of prefix moves that are required to get the nearest free slot from above (below) the block where it is needed and *getFromAbove* and *getFromBelow*, respectively, get the nearest free slot above or below the block where the free slot is needed.

The algorithm to free a slot (Figure 1.25) adds a freed slot inside the block to the doubly-linked list of free slots. Again, correctness and consistency are established easily.

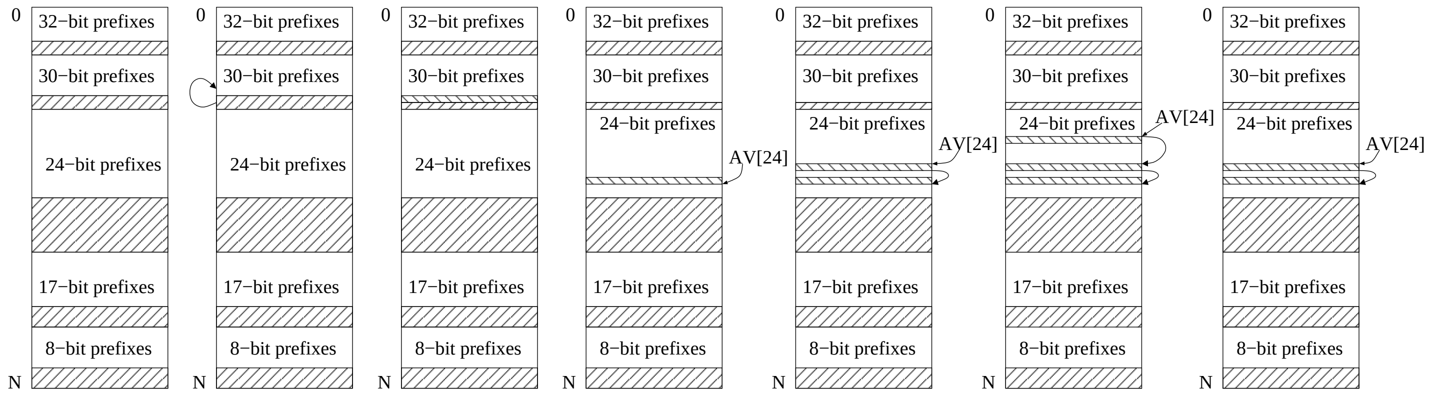
The scheme is shown in Figure 1.28(a), where the ITCAM slots are indexed 0 through N. The prefixes are stored in decreasing order of length in the TCAM, which ensures that the longest matching prefix is returned as the first matching prefix. Prefixes of the same length appear one after another in a group in the TCAM and this is referred to as a prefix block. The free slots are distributed between the boundaries of adjacent prefix blocks. At the time the ITCAM is initialized, the available free slots are distributed in proportion to the number of prefixes in a block with the caveat that an empty block gets one free slot at its boundary. In addition to the free space at the boundary regions, a doubly-linked list of free slots is maintained within each block. Free slots within a block are created as a result of prefix deletions from within the block.

```
Algorithm: movesFromAbove(len, *pos, *cur)
    moves=0;
    // find max p ≤len such that block p is not empty
    for (p=len; top[p] > bot[p]; p−−);
    // find min c > len with space just below it
    for (c=len+1; c≤W+1; c++)
        if (top[c] ≤ bot[c]) // not empty
            if (bot[c]+1 < top[p] || !valid[bot[c]])
                *cur = c; *pos = p;
                return moves;
            endif
            moves++;
            if (AV[c] >= 0) *pos = c; *cur = c; return moves; endif
            p = c;
        endif
    return W + 1;

Algorithm: movesFromBelow(len, *pos, *cur)
    moves=0;
    // find min p >= len such that block p is not empty
    for (p=len; top[p] > bot[p]; p++);
    // find min c > len with space just below it
    for (c=len-1; c>=0; c−−)
        if (top[c] ≤ bot[c]) // not empty
            if (top[c]-1 > bot[p] || !valid[top[c]])
                *pos = p; *cur = c;
                return moves;
            endif
            moves++;
            if (AV[c] >= 0) *pos = c; *cur = c; return moves; endif
            p = c;
        endif

    return W + 1;
```

FIGURE 1.26: Algorithms to compute the number of moves used by the algorithm of Figure 1.24

```
Algorithm: getFromAbove($len$, $p$, $c$)
    if (top[$p$] $>$ bot[$c$]+1) $d$ = top[$p$]-1; $c = p$;
    else
        if (!valid[bot[$c$]])
            $d$ = bot[$c$]−−;
            if ($d$ == AV[$c$]) AV[$c$] = next[AV[$c$]];
            else
                next[prev[$d$]] = next[$d$];
                if (next[$d$] != -1) prev[next[$d$]] = prev[$d$];
            endif
        else
            $d$ = AV[$c$];
            AV[$c$] = next[AV[$c$]];
            $move$(bot[$c$], $d$);
            $d$ = bot[$c$]−−;
        endif
        $c - -$;
    endif
    for (; $c > len$; $c - -$)
        if (top[$c$] $\leq$ bot[$c$])
            $move$(bot[$c$]−−, −−top[$c$]);
            $d$ = bot[$c$]+1;
        endif
    return $d$;

Algorithm: getFromBelow($len$, $p$, $c$)
    if (top[$c$]-1 $>$ bot[$p$]) $d$ = bot[$p$]+1; $c = p$;
    else
        if (!valid[top[$c$]])
            $d$ = top[$c$]++;
            if (d == AV[$c$]) AV[$c$] = next[AV[$c$]];
            else
                next[prev[$d$]] = next[$d$];
                if (next[$d$] != -1) prev[next[$d$]] = prev[$d$];
            endif
        else
            $d$ = AV[$c$];
            AV[$c$] = next[AV[$c$]];
            $move$(top[$c$],$d$);
            $d$ = top[$c$]++;
        endif
        $c$++;
    endif
    for (; $c < len$; $c$++)
        if (top[$c$] $\leq$ bot[$c$])
            $move$(top[$c$]++, ++bot[$c$]);
            $d$ = top[$c$]-1;
        endif
    return $d$;
```

FIGURE 1.27: Algorithms to get a slot used by the algorithm of Figure 1.24



(a) Initial arrangement (b) Insert p/30 (c) Free space available (d) Delete p/24 (e) Delete p2/24 (f) Delete p3/24 (g) Insert p/24

FIGURE 1.28: ITCAM layout for DLFS_PLO, with moves for insert and delete. The curved arrows on the right show the forward links in the list of free spaces.

The number of moves required by an update sequence is dependent on the number of free TCAM slots available. An experiment in [6] shows that even with 99% prefix occupancy and 1% free space, the total number of prefix moves using DLFS_PLO is at most 0.7% of the total number of prefix inserts and deletes.

1.5.6 DUOW

DUOW is a dual TCAM architecture in which a wide SRAM (say, 144-bit words or larger) is used with either or both the TCAMs. The method of adding a wide LSRAM to the LTCAM is described here. The technique of adding a wide ISRAM is almost identical to that used in Section 1.4. A wide LSRAM word is used to store a subtree of the binary trie of a forwarding table. Since the LTCAM stores prefixes in the leaf nodes of a binary trie, carving is done on a leaf trie storing the leaf prefixes only and not on the original binary trie storing all prefixes. When a subtree of the leaf trie is stored in an LSRAM word, that subtree is removed from (or carved out of) the leaf trie before another subtree is identified for carving. Let N be the root of the subtree being carved and let $Q(N)$ be the prefix defined by the path from the root of the trie to N. $Q(N)$ is stored in the LTCAM, and $|P_i| - |Q(N)|$ suffix bits, of each prefix P_i in the carved subtree rooted at N, are stored in the LSRAM word. Note that each suffix stored in the LSRAM word is a suffix of a leaf prefix that begins with $Q(N)$. By repeating this carving process, all leaf prefixes are allocated to the LTCAM and LSRAM. To obtain the mapping of leaf prefixes to the LTCAM and LSRAM, the carving algorithm must ensure that the $Q(N)$s stored in the LTCAM are disjoint. Since the carving algorithm in Section 1.4 does not ensure disjointedness, a new carving algorithm is needed. As an example, consider the binary trie of Figure 1.29(a), which has been carved using a carving algorithm that ensures that each carved subtree has at most 2 leaf prefixes. The LTCAM will need to store $Q(N1)$, $Q(N2)$ and $Q(N3)$. Even though the prefixes in the binary trie are disjoint, the $Q(N)$s in the LTCAM are not disjoint (e.g., $Q(N1)$ is a descendant of $Q(N2)$ and so $Q(N2)$ matches all IP addresses matched by $Q(N1)$). To retain much of the simplicity of the LTCAM management scheme of DUOS it is necessary to carve the leaf trie in such a way that all $Q(N)$s in the LTCAM are disjoint.

In this case, carving is done via a postorder traversal of the binary trie, using the visit algorithm of Figure 1.30 to do the carving. In this algorithm, w is the number of bits in an LSRAM word and $x \rightarrow$size is the number of bits needed to store (1) the suffix bits corresponding to prefixes in the subtrie rooted at x, (2) the length of each suffix, (3) the next hop for each suffix, (4) the number of suffixes in the word, and (5) the length of $Q(x)$, which is the corresponding prefix stored in the LTCAM. Algorithm $splitNode(q)$ does the actual carving of the subtree rooted at node q, which involves setting child pointers appropriately for the parent of the carved node, as well as adjusting the number of prefixes and the SRAM bits needed for the rest of the trie in the

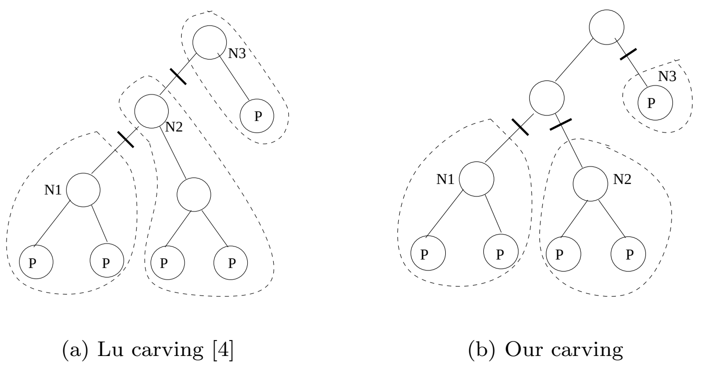


FIGURE 1.29: Carving using the method of [4] and our method

```
Algorithm: visit_postorder($x$)
    if (!$x$) return 0;
    isSplit = visit_postorder($y$);
    isSplit |= visit_postorder($z$);
    if (isSplit ||$x$→size > w) then
        splitNode($y$);
        splitNode($z$); // where $y$ and $z$ are children of $x$
        return 1;
    else if ($x$→size == w) then
        splitNode($x$);
        return 1;
    endif
    return 0;
```

FIGURE 1.30: Algorithm to carve a leaf trie to obtain disjoint $Q(N)$s

absence of the subtree being carved. The basic idea in our carving algorithm is to forbid carving at two nodes that have an ancestor-descendent relationship. This ensures that the $Q(N)$s are disjoint. Figure 1.29(b) shows the subtrees carved by our algorithm. As can be seen, $Q(N1)$, $Q(N2)$, $Q(N3)$ are disjoint. Although our carving algorithm generally results in more $Q(N)$s than when the carving algorithm of [4] is used, our carving algorithm allows us to retain the flexibility to store the $Q(N)$s in any order in the LTCAM as the $Q(N)$s are independent.

The LTCAM algorithms to insert, delete, change, and necessary support algorithms are given in [6]. Figure 1.31 shows a possible assignment of the 6-prefix example in Figure 1.1. The suffix node format need not store the default next hop, since in case of a "no match" situation the ITCAM provides the next hop. The intermediate prefixes for rules R1, R3 and R4 are stored in the ITCAM, while the leaf prefixes for rules R2, R5 and R6 are stored in the LTCAM using a wide LSRAM. The suffix nodes begin with the prefix length field of 2 bits in this example followed by the suffix count field of 2 bits. Next comes the (length, suffix, nexthop) triplet for each prefix encoded in the suffix node, the number of allocated bits being (2bits, 4 bits, 6 bits) respectively for the three fields in the triplet.

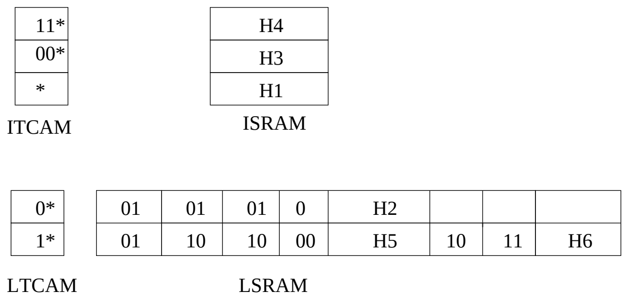


FIGURE 1.31: Assignment of prefixes of Figure 1.1 to DUOW

1.5.7 IDUOW

It is evident that by adding an index TCAM in conjunction with an LTCAM that uses a wide LSRAM (i.e., an index for the LTCAM of DUOW), there was further reduction in power consumption. Adding an index to the ITCAM of DUOW follows easily from [4]. When the LTCAM is indexed, we have two TCAMs replacing the LTCAM – a data TCAM referred to as DLTCAM and an index TCAM referred to as ILTCAM. The associated SRAMs are DLSRAM and ILSRAM.

Two index TCAM strategies based on the 1-12Wc and M-12Wb schemes of [4] were explored in [6]. The 1-12Wc scheme is best for power whereas the M-12Wb scheme is the best overall scheme consuming least TCAM space and low power for lookups [4]. Both 1-12Wc and M-12Wb organize the DLTCAM into fixed size buckets that are indexed using the ILTCAM and ILSRAM, which also is a wide SRAM that stores suffixes and associated information.

The memory management scheme specifies how a new bucket is assigned when a prefix is to be added to a bucket that is already full. Similarly, as a prefix is deleted from a bucket, the memory management scheme explores the possibility of merging the contents of the bucket with another bucket and updating the DLTCAM and ILTCAM contents accordingly.

The algorithms for carving and prefix assignment for the 1-12Wc and M-12Wb prefix layout schemes as well as the memory management algorithms for both schemes can be found in [6].

1.6 Experimental Results

The performance evaluation of the different TCAM schemes are presented in [6] and are summarized here for 20 IPv4 routing tables and update sequences downloaded from [17] and [18]. Figure 1.32 gives the characteristics of these datasets. The update sequences for the first 20 routing tables were captured from files storing update announcements from 12am on February 1, 2009 for the stated number of hours. The columns labeled $\#RawInserts$,

DataSet	#Prefixes	Hours	$\#RawInserts$	$\#RawDeletes$	$\#RawChanges$
rrc00	294098	75.7	39553	40051	368013
rrc01	276795	75.2	41692	41988	492315
rrc03	283754	42.7	27702	27914	292454
rrc04	288610	17	16086	15977	193392
rrc05	280041	103	20276	18285	439647
rrc06	278744	235	157549	157547	289272
rrc07	275097	0.417	247	218	179835
rrc10	278898	105	21620	22473	326720
rrc11	277166	80.2	58115	58378	290621
rrc12	278499	62.3	33196	33572	410464
rrc13	284986	57.8	23920	23713	284710
rrc14	276170	83.6	56598	56810	203955
rrc15	284047	134	95790	93750	183131
rrc16	282660	672	3338	937	8896
rv2	294127	56.5	13882	15552	679100
rv4	275737	95	69627	69754	526302
rv.eqix	275736	70.3	51104	51066	253693
rv.isc	281095	68.2	44286	44444	292323
rv.linx	278196	49.1	23137	23413	384344
rv.wide	283569	174	101821	103862	372035

FIGURE 1.32: Datasets used in the experiments

$\#RawDeletes$ and $\#RawChanges$, respectively, give the number of insert, delete, and change next hop requests in the update sequences. Using consistent updates, a next hop change request is implemented (see Figure 1.20 for example) as an insert (of the prefix with the new next hop) followed by a delete (of the prefix with the old nexthop). Therefore, all results henceforth are in terms of the effective inserts and deletes. Note that the number of effective inserts (#Inserts) and deletes (#Deletes) is given by the following equations.

$$\#Inserts = \#RawInserts + \#RawChanges; \tag{1.3}$$

$$\#Deletes = \#RawDeletes + \#RawChanges; \tag{1.4}$$

Figures 1.33 and 1.34 show a comparison of the TCAM and SRAM power consumption among the TCAM schemes that support incremental updates.

The IDUOW data assume that the LTCAM is indexed using an ILTCAM but the ITCAM is not indexed. The STCAM in Section 1.2 can be updated incrementally by using DLFS_PLO. The power numbers were estimated using CACTI [20] for SRAMs and "tcam-power" [21] for TCAMs. Additionally, the following settings were used: process technology of 70nm, operating voltage of 1.12V, TCAM operation frequency of 360MHz [22]. For the IDUOW schemes, the bucket size for DLTCAM was set to 512 entries.

The IDUOW schemes reduce TCAM power by an order of magnitude relative to the STCAM scheme and DUOS. For both the IDUOW schemes, the ILTCAM and DLTCAM together consume less than 70mW; the ITCAM consumes the remaining approximately 2W. When the ILTCAM is also indexed using its own index TCAM, the total power is less than 100mW, which is a 200 fold reduction relative to STCAM and DUOS.

DataSet	STCAM	DUOS	DUOW	IDUOW-(1-12Wc)	IDUOW-(M-12Wb)
rrc00	24246.29	24246.43	7880.15	2303.91	2308.04
rrc01	22820.04	22820.18	7375.72	2090.03	2093.65
rrc03	23393.68	23394.24	7600.31	2199.94	2203.34
rrc04	23794.03	23794.59	7623.57	2119.8	2124.49
rrc05	23087.36	23087.92	7478.02	2138.96	2143.34
rrc06	22980.46	22981.02	7438.55	2124.73	2130.96
rrc07	22679.9	22680.46	7318.64	2061.66	2066.36
rrc10	22993.33	22993.47	7422.78	2093.83	2097.4
rrc11	22850.44	22850.57	7383.31	2090.03	2094.57
rrc12	22960.24	22960.8	7422.87	2099.1	2102.59
rrc13	23495.32	23495.46	7641.93	2216.5	2221.2
rrc14	22768.28	22768.84	7343.06	2065.21	2068.7
rrc15	23417.77	23418.33	7599.08	2205.51	2211.34
rrc16	23303.6	23303.74	7539.56	2155.93	2160.39
rv2	24248.63	24249.2	7847.89	2263.41	2268.03
rv4	22732.87	22733.01	7336.02	2069.9	2075.65
rv.eqix	22732.79	22732.93	7328.82	2060.2	2065.3
rv.isc	23174.63	23174.76	7513.12	2145.59	2149.32
rv.linx	22935.34	22935.48	7422.61	2113.56	2119.55
rv.wide	23378.31	23378.86	7611.94	2224.35	2230.26

FIGURE 1.33: TCAM power consumption in mW

Incremental updates are efficiently performed in DUO. The DLFS_PLO memory management scheme used in DUO requires very few prefix moves, as can be seen in Figure 1.35 which gives the total and average number of prefix moves as well as the standard deviation from the average, while applying the inserts and deletes on the STCAM. The average number of prefixes moved is obtained by dividing the total moves by the total number of inserts and deletes performed on the forwarding table.

The total number of *waitWrite*s for the inserts and deletes are presented in the Figure 1.36. As noted in [6], the maximum number of *waitWrite*s for the

DataSet	STCAM	DUOS	DUOW	IDUOW-(1-12Wc)	IDUOW-(M-12Wb)
rrc00	1412.07	1427.18	1639.49	170.98	173.78
rrc01	1329.46	1343.17	1549.85	157.49	159.91
rrc03	1362.26	1376.69	1585.82	164.29	166.6
rrc04	1385.34	1399.77	1608.5	159.77	162.9
rrc05	1345.26	1359.51	1566.66	160.58	163.53
rrc06	1339.18	1352.6	1558.92	159.59	163.76
rrc07	1322.17	1335.27	1539.72	155.66	158.83
rrc10	1339.18	1353.05	1561.08	157.76	160.18
rrc11	1331.88	1345.6	1551.24	157.49	160.55
rrc12	1337.96	1352.17	1560.03	157.98	160.34
rrc13	1368.34	1382.72	1593.89	165.5	168.67
rrc14	1327.03	1340.29	1545.43	155.83	158.19
rrc15	1364.69	1378.24	1584.76	164.67	168.58
rrc16	1357.4	1371.62	1579.02	161.69	164.7
rv2	1412.07	1426.95	1638.21	168.43	171.55
rv4	1324.6	1338.19	1542.99	156.16	160.01
rv.eqix	1324.6	1338.75	1542.33	155.5	158.94
rv.isc	1350.11	1364.87	1574.2	160.97	163.5
rv.linx	1336.75	1350.73	1556.88	159.03	163.05
rv.wide	1362.26	1376.97	1584.54	165.94	169.9

FIGURE 1.34: SRAM power consumption in mW

Dataset	Total moves		Average moves		Standard Deviation	
	insert	delete	insert	delete	insert	delete
rrc00	401	0	0.000984	0	0.064	0
rrc01	0	0	0	0	0	0
rrc03	4630	0	0.0145	0	0.311	0
rrc04	2	0	0.00001	0	0.003	0
rrc05	541	0	0.00118	0	0.077	0
rrc06	8	0	0.00002	0	0.004	0
rrc07	0	0	0	0	0	0
rrc10	671	0	0.00193	0	0.069	0
rrc11	245	0	0.0007	0	0.0377	0
rrc12	4659	0	0.0105	0	0.27	0
rrc13	989	0	0.0032	0	0.114	0
rrc14	4	0	0.000015	0	0.005	0
rrc15	2769	0	0.0099	0	0.212	0
rrc16	0	0	0	0	0	0
rv2	14	0	0.00002	0	0.005	0
rv4	141	0	0.000236	0	0.034	0
rv.eqix	33	0	0.000108	0	0.0107	0
rv.isc	12	0	0.000036	0	0.006	0
rv.linx	0	0	0	0	0	0
rv.wide	1	0	0.000002	0	0.0015	0

FIGURE 1.35: Statistics on the number of prefix moves

IDUOW scheme using M-12Wb layout is quite large, but it is easily reducible by fixing the size of a DLTCAM bucket to a smaller number.

Dataset	DUOS	DUOW	IDUOW(1-12Wc)	IDUOW(M-12Wb)
rrc00	831630	894978	904692	923390
rrc01	1083395	1151102	1160645	1179951
rrc03	655262	703419	711292	726559
rrc04	425587	453943	458190	477705
rrc05	924947	958053	961356	987024
rrc06	950924	1194841	1224466	1254236
rrc07	360149	360588	360596	362235
rrc10	703412	746524	749215	769919
rrc11	715786	809353	819181	844235
rrc12	908971	964395	975039	993097
rrc13	630400	660925	666808	691469
rrc14	536465	632699	640566	658301
rrc15	585012	720082	735449	767875
rrc16	22919	26107	28480	52885
rv2	1397932	1419396	1424525	1443112
rv4	1225347	1352059	1368462	1400013
rv.eqix	624700	705100	712971	742635
rv.isc	695496	767463	778565	797057
rv.linx	826755	861534	868559	897055
rv.wide	988721	1145004	1163932	1195740

FIGURE 1.36: Total number of TCAM $waitWrite$ operations

The worst case number of writes needed for an insert or delete in DUOS is 34, in DUOW is 38, and in IDUOW $38+bucketSize$.

1.7 Conclusion

The techniques of indexing a TCAM and the use of wide SRAMs were explored as mechanisms for reducing the power consumed by TCAM-based routers. It was found that the techniques when applied together reduced power consumption by more than two orders of magnitude. Using the dual TCAM architecture, DUO, for forwarding tables it was possible to have the benefits of low power consumption along with efficient incremental updates and faster TCAM lookup. The absence of a priority encoder in the LTCAM system of DUO made overall lookup faster by about 50% compared to a conventional TCAM, when the nexthop was found in the LTCAM. It was observed that over 90% prefixes are stored in the leaf TCAM system. Therefore, the probability of having a hit in the LTCAM system during lookup is very high. The lookup consistent algorithms for incremental updates allowed data plane lookup to proceed along with the control plane update operations.

1.8 Acknowledgement

This work was supported, in part, by the National Science Foundation, under grant 0829916 and the US Air Force under grant FA8750-10-1-0236.

Bibliography

[1] R. S. Tucker, J. Baliga, R. Ayre, K. Hinton, and W. V. Sorin, Energy Consumption in IP Networks, *34th European Conference and Exhibition on Optical Communication*, Sept. 2008.

[2] A. M. Lyons, D. T. Neilson, and T. R. Salamon, Energy efficient strategies for high density telecom applications, Workshop On Wireless Communications and Networks, Feb. 2008.

[3] F. Zane, G. Narlikar and A. Basu, CoolCAMs: Power-Efficient TCAMs for Forwarding Engines, *INFOCOM*, 2003.

[4] W. Lu and S. Sahni, Low Power TCAMs For Very Large Forwarding Tables, *Proceedings of INFOCOM*, 2008.

[5] M. Akhbarizadeh and M. Nourani, Efficient Prefix Cache for Network Processors, *IEEE Symposium on High Performance Interconnects*, August 2004.

[6] T. Mishra and S.Sahni, DUO-Dual TCAM Architecture for Routing Tables with Incremental Update, *http://www.cise.ufl.edu/˜sahni/papers/duo.pdf*

[7] H. Liu, Routing Table Compaction in Ternary-CAM, *IEEE Micro*, 22, 3, 2002.

[8] T. Mishra and S.Sahni, PETCAM – A Power Efficient TCAM Architecture For Forwarding Tables, *IEEE Transactions on Computers*, 2010

[9] C. A. Zukowski, and S. Wang, Use of Selective Precharge for Low-Power Content-Addressable Memories, *IEEE International Symposium on Circuits and Systems*, 1997.

[10] N. Mohan, and M. Sachdev, Low Power Dual Matchline Ternary Content Addressable Memory, *IEEE International Symposium on Circuits and Systems*, 2004.

[11] H. Miyatake, M. Tanaka, and Y.Mori, A design for high-speed low-power CMOS fully parallel content addressable memory macros, *IEEE Journal of Solid State Circuits*, 36, 6, June 2001, 956-968.

[12] C.-S. Lin, J.-C. Chang, and B.-D Liu, A low-power pre-computation based fully parallel content addressable memory, *IEEE Journal of Solid State Circuits*, 38, 4, April 2003, 654-662.

[13] R. Draves, C. King, S. Venkatachary, and B.Zill, Constructing Optimal IP Routing Tables, *Proceedings of INFOCOM*, 1999, 88-97.

[14] G. Wang and N. Tzeng TCAM-Based Forwarding Engine with Minimum Independent Prefix Set (MIPS) for Fast Updating, *IEEE International Conference of Communications* Volume 1, June 2006, 103-109

[15] Z. Wang, H. Che, M. Kumar, and S.K. Das, CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking, *IEEE Transactions on Computers*, 53, 12, December 2004, 1602-1614.

[16] D. Shah and P. Gupta, Fast Updating Algorithms on TCAMs, *IEEE Micro* Volume 21, Issue 1, Jan-Feb 2001, 36-47

[17] *http://www.routeviews.org*, 2009.

[18] *http://www.ripe.net/projects/ris/rawdata.html*, 2008.

[19] Satoshi Sekiguchi, Grid around Asia, *Grid Asia*, 2009

[20] N. Muralimanohar, R. Balasubramonian and N. P. Jouppi, Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0, *Proceedings of the 40th International Symposium on Microarchitecture* December 2007, 3-14

[21] B. Agrawal and T. Sherwood, Ternary CAM Power and Delay Model: Extensions and Uses, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 16, No. 5, May 2008, 554-564.

[22] Renesas R8A20410BG 20Mb Quad Search Full Ternary CAM. http://www.renesas.com/products/memory/TCAM/tcam_root.jsp. January 2010.