IMT: A Memory-Efficient and Fast Updatable IP Lookup Architecture Using an Indexed Multibit Trie

Junghwan Kim[1], Myeong-Cheol Ko[1], Moon Sun Shin[1] and Jinsoo Kim[1*]
[1] Department of Computer Engineering, Konkuk University
268 Chungwon-daero, Chungju-si, Chungcheongbuk-do, 27478, Korea
[e-mail: {jhkim, cheol, msshin, jinsoo}@kku.ac.kr]
*Corresponding author: Jinsoo Kim

Received August 2, 2017; revised May 21, 2018; revised August 23, 2018; accepted November 10, 2018; published April 30, 2019

Abstract

IP address lookup is a function to determine nexthop for a given destination IP address. It takes an important role in modern routers because of its computation time and increasing Internet traffic. TCAM-based IP lookup approaches can exploit the capability of parallel searching but have a limitation of its size due to latency, power consumption, updatability, and cost. On the other hand, multibit trie-based approaches use SRAM which has relatively low power consumption and cost. They reduce the number of memory accesses required for each lookup, but it still needs several accesses. Moreover, the memory efficiency and updatability are proportional to the number of memory accesses. In this paper, we propose a novel architecture using an Indexed Multibit Trie (IMT) which is based on combined TCAM and SRAM. In the proposed architecture, each lookup takes at most two memory accesses. We present how the IMT is constructed so as to be memory-efficient and fast updatable. Experiment results with real-world forwarding tables show that our scheme achieves good memory efficiency as well as fast updatability.

Keywords: IP address lookup, indexed multibit trie, subtrie-pushing, TCAM-based index, prefix updatability

1. Introduction

Internet traffic has been rapidly increased for past several decades and demanded more powerful forwarding engine. IP address lookup is one of the key functions required for forwarding packets in a router. It is to determine the nexthop information for a given destination IP address. The IP address lookup consumes much more computational cycles in these days because it requires Longest Prefix Matching (LPM) due to Classless Inter-Domain Routing (CIDR) [1]. The growth of routing information also demands high-performance IP lookup engine. The recent routing table contains more than 600,000 network prefixes.

Many IP address lookup schemes have been researched that generally fall into either of two categories: trie-based and TCAM-based. In trie-based approach, it often requires multiple accesses to the trie structure. So most of research focused on reducing the number of memory accesses [2]. Multibit trie is one of the basic enhancements to reduce memory accesses. A prefix can be expanded to a longer node using called prefix expansion technique in a multibit trie [3]. Multibit trie-based approaches proceed to search with a long stride unlike a binary trie (or unibit trie).

Ternary Content-Addressable Memory (TCAM) is a specialized memory that searches the entire entries in a single cycle in parallel. Each cell can represent * as well as 0 or 1, so it is suitable for storing prefixes. It can search for the longest matching prefix if priority-based selection logic is provided. However, its size is restrictive because it consumes more power and takes longer latency than SRAM. The power consumption is reduced by routing table compaction and partitioning [4]. Furthermore, hybrid architectures of TCAM-SRAM were proposed to exploit both advantages [5][6].

The update overhead is one of the major issues in high-performance IP lookup engine. The update time may affect lookup speed because it consumes computational resources and may lock the lookup operation during the update. The number of memory entries to store a single prefix affects the update time. Longer stride in a multibit trie makes it possible to lookup fast, but it takes longer update time because more entries are often associated with a single prefix.

In this paper, we propose a new novel IP address lookup architecture which especially focuses on memory efficiency and fast updatability. In that architecture, we use both TCAM and SRAM to store multibit trie-based lookup structure. Unlike the conventional multibit trie our scheme performs lookups very fast only requiring at most two memory accesses. The size of each subtrie is controlled to satisfy memory efficiency and updatability criteria. An elaborate technique is also devised to control the number of TCAM accesses incurred by a single update.

The rest of this paper is organized as follows. Section 2 reviews related works, and Section 3 explains conventional IP lookup methods employing multibit trie and presents our motivation. Section 4 describes our proposed architecture with a lookup scheme, construction algorithm for IMT, and an update algorithm. Experiment results are presented and discussed in Section 5. Lastly, Section 6 concludes the paper.

2. Related Works

There has been much research on power consumption of TCAM [4][7-9]. In CoolCAMs, the routing table is partitioned into sub-tables (TCAM buckets) to reduce power consumption [7]. Lu and Sahni made enhanced CoolCAMs and proposed several architectures based on the use of wide SRAM [8]. Reduction of the number of entries in a TCAM-based routing table is crucial in point of power consumption, cost, and lookup delay. Liu suggested some techniques to reduce the entries [4].

Several TCAM-based IP lookup engines have been proposed. Akhbarizadeh *et al*. proposed a TCAM-based IP forwarding engine using prefix segregation scheme [10]. In that scheme, prefixes in the routing table are partitioned into two groups. One consists of non-overlapped prefixes, and the other group covers remaining prefixes. These two groups are contained in two separate TCAMs. The TCAM containing non-overlapped prefixes does not need priority encoder and sorting the prefixes in order of length, so it achieves good lookup performance and fast updates. Akhbarizadeh *et al.* partitioned prefixes and stored them into several TCAM blocks as well [11]. In this scheme, multiple selectors deliver incoming traffic to those blocks in parallel. It exploits popular prefixes stored in early stage memory to reduce contention in TCAM blocks and enhance the lookup performance.

In TCAM-based lookup schemes, all entries should be sorted to guarantee the longest prefix matching, which makes it difficult to update incrementally. Shah and Gupta proposed a novel prefix ordering scheme, chain-ancestor order, which reduces the number of memory movements on updates [12].

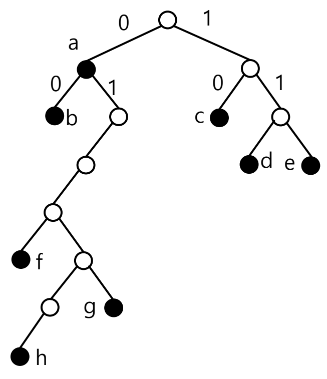
Luo *et al*. proposed a hybrid architecture which consists of TCAM-based lookup engine and SRAM-based pipelined engine [5]. In that architecture, FIB is partitioned into two sets. Disjoint leaf prefixes are mapped into TCAM-based engine and other overlapping prefixes are mapped into SRAM-based engine to increase throughput and to reduce the update overhead. Kim *et al*. proposed a hybrid architecture based on SRAM and TCAM [6]. In that architecture, 16-bit indices are stored in SRAM while prefixes longer than 16 bits are distributed over TCAM blocks. Since a TCAM block is selectively activated by an index, high throughput is achieved with low power. Prefixes are evenly distributed over TCAM blocks by elaborated technique so the size of TCAM is minimized in point of entries and width.

Several trie-based or multibit trie-based schemes have been proposed [3][13-20]. Srinivasan and Varghese proposed leaf pushing and multibit trie [3]. Le at al. devised an SRAM-based bidirectional pipeline for high throughput IP lookup engine [13]. In that scheme, leaf-pushed unibit trie is used for IP address lookup. They tried to optimize memory balancing which is crucial in the pipelined architecture. Basu and Narlikar designed an SRAM-based pipeline to process IP lookup [14]. In the pipeline, a leaf-pushed trie is allocated to the stages and balancing the allocation is crucial. Write bubble technique was proposed to handle updates in the pipeline efficiently. Lee and Lim proposed multi-stride decision trie which satisfies some characteristics [15]. Wu et al. proposed pipelined IP lookup scheme based on a prefix trie [16]. In that scheme, prefixes are classified into short and long groups. The short group is contained in a quick table and the long group is indexed by the quick table.

Chu *et al*. proposed a GPU-based parallel architecture [17]. A novel variable-stride multibit trie-based data structure was devised to reduce search steps and to optimize memory accesses in GPU. Li *et al.* devised a fixed-stride multibit trie to compact the data structure and utilize the GPU’s memory access pattern fully [18]. In that scheme, each unit is either a nexthop or a pointer to a child by means of leaf-pushing, i.e., it is a disjoint multibit trie. Li *et al.* used a multibit trie-based table and pursued the efficiency of on-chip memory containing the lookup table by splitting the table [21]. However, for off-chip access, they should use hashing and it needs parallel memory accesses or sparse slots to avoid conflicts. It is assumed each traversing step in the multibit trie is pipelined and the split table is accessed in parallel.

3. Conventional Multibit Tries for IP lookup

This section describes in detail the background knowledge about the multibit trie that has been previously developed. A trie is a kind of search tree structure used to find the Longest Matching Prefix (LMP) for a destination IP address. A *binary trie* (or *unibit trie*) is the most fundamental trie whose degree is two [2]. A node in a binary trie is represented by a bit string which indicates the path from the root to that node. **Fig. 1(a)** shows an instance of a binary trie in which prefixes are represented by filled circles (*a* ~ *h*). The LMP is the last visited prefix when a trie is traversed from the root node. For example, given a destination address 0100100 (7-bit long for simplicity), the LMP should be the prefix *h* though the prefix *a* is also matched. In a binary trie, the IP lookup time is accounted for by the number of accessed nodes. The number of memory accesses for each prefix is shown in **Fig. 1(b)**. In the worst case, it is the trie depth, which is 32 in IPv4 and 128 in IPv6.



(a) A binary trie with eight prefixes

	Prefix	Mem. acc. in BT[1)]	Mem. acc. in MT[2)]
a	0*	2	1
b	00*	3	1
c	10*	3	1
d	110*	4	2
e	111*	4	2
f	01000*	6	2
g	010011*	7	3
h	0100100*	8	3
Avg. mem. acc.		4.63	1.88

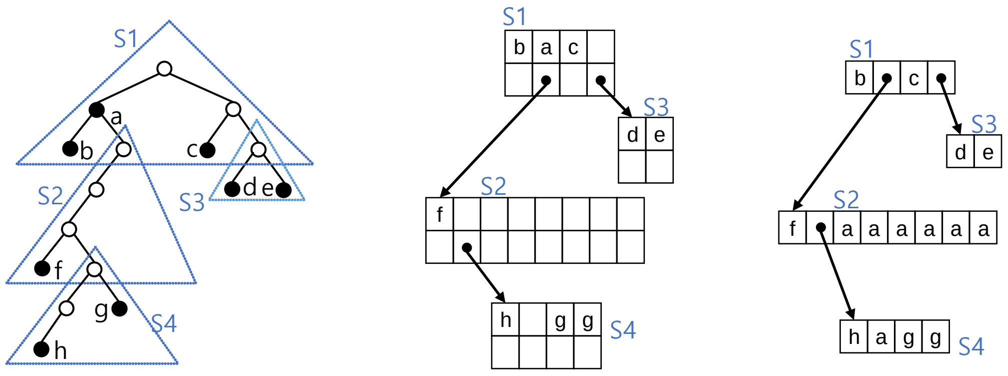
[1)] BT = Binary Trie
[2)] MT = Multibit Trie

(b) The number of memory accesses

Fig. 1. An example of prefixes and the corresponding memory accesses

The number of memory accesses can be reduced if several bits are checked in a node. In a *multibit trie*, several bits can be checked at once in a node [3]. The number of bits to be checked per node is called *stride*. **Fig. 2** shows a multibit trie corresponding to **Fig. 1(a)**. In the figure, each of S1 ~ S4 represents a multibit trie node and we call it a *subtrie* of a multibit trie. **Fig. 2(a)** depicts a conceptual view in which each subtrie range is projected to the binary trie. **Fig. 2(b)** shows the actual memory state for the multibit trie. In that figure, each entry consists of the nexthop associated with a prefix and a child pointer. In **Fig. 2(c)**, prefix *a* is expanded to leaves by a technique called *leaf-pushing* [3]. In a leaf-pushed multibit trie, all the prefixes are located at leaves and matching always occurs at leaves. Note that each entry contains either the nexthop or a child pointer but not both in a leaf-pushed multibit trie. So a leaf-pushed multibit trie requires less memory space.

Compared to a binary trie, a multibit trie requires less memory accesses for each lookup. In **Fig. 1(b)**, the average number of memory accesses of the multibit trie is 1.88 while that of the binary trie is 4.63. However, there are some disadvantages on a multibit trie. First, it still requires several memory accesses and even the number of memory accesses is considerably high according to the depth of the multibit trie. In **Fig. 2**, it requires at least three memory accesses to match the prefix *g* or *h*. Second, while longer stride makes the number of memory accesses decrease, it causes poor memory efficiency. In **Fig. 2(b)**, among 8 entries only one entry contains a prefix in S2 when the stride is 3. Also, long stride derives too many entries per prefix, so it may experience poor updatability. Third, leaf-pushing causes a lot of memory accesses for a single prefix update. For example, in **Fig. 2(c)**, when the prefix *a* is updated, all the leaf-pushed 7 entries should be accessed even in different subtries.



(a) Conceptual view (b) Basic multibit trie (c) Leaf-pushed multibit trie

Fig. 2. Two types of multibit tries

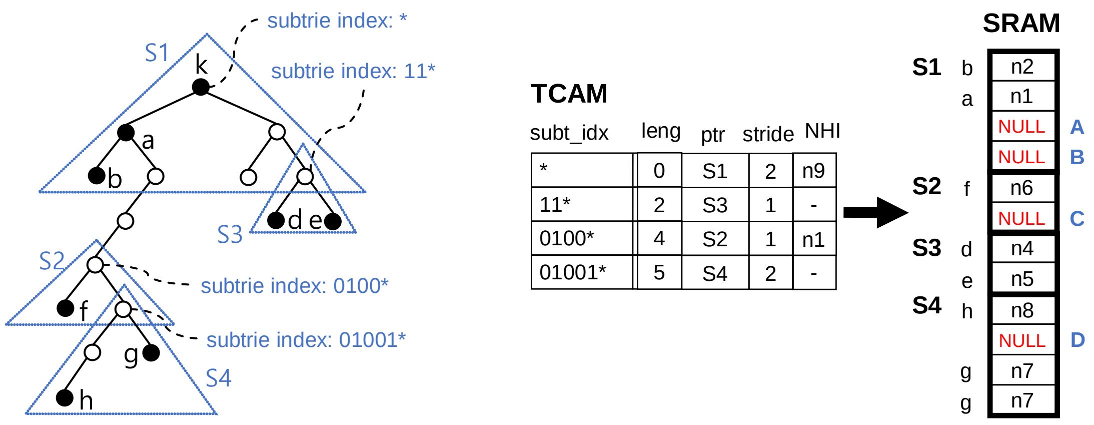
4. IP Address Lookup Using an Indexed Multibit Trie

4.1 Overall Scheme

The prefix expansion in multibit tries considerably reduces the number of steps to reach the longest matching prefix compared to binary tries. However, all the subtries on the traversed path have to be visited, so quite a time is still taken in matching process if the depth of the trie is high. We propose a new scheme to achieve fast lookup without visiting intermediate subtries. In this scheme, only the last matching subtrie is directly visited using an index without visiting intermediate subtries.

Fig. 3 shows an *Indexed Multibit Trie* (*IMT*) which is constructed by our scheme. The construction algorithm will be described in Section 4.4. Unlike **Fig. 2**, each subtrie is independent and directly accessible without going through the intermediate subtries in the multibit trie of **Fig. 3**. All subtries are stored in SRAM and each entry is accessed using direct addressing as the conventional multibit tries. To determine a subtrie, we use the *subtrie index* which is the root of that subtrie. Since the lengths of subtrie roots are various, we exploit TCAM to store and search those values. **Fig. 3 (b)** shows the subtrie indexes in TCAM and all the subtries in SRAM. IP address lookup can be performed very fast irrespective of the length of the matched prefix since it is always completed by accesses to TCAM and SRAM. Let us consider a straightforward example. Given an input IP address, 0100110 (7-bit address for simplicity), the longest matching prefix *g* will be found in two steps without going through the subtrie S1 and S2. First, the TCAM is searched with 0100110, and the fourth entry is returned as the longest matching entry though the first and the third entries are also matched. Since the length of S4's root is 5, the 5 significant bits of 0100110 are masked. Then, next 2 bits are used as offset because stride is 2. Now, the SRAM is accessed using the subtrie pointer S4 and the offset of 2 (= 10_2).

Each SRAM entry only has *Next Hop Identifier* (*NHI*) field and the field for the pointer to a child subtrie is not required because the intermediate subtries along the path need not to be actually traversed. So the capacity of required SRAM is almost as half as those of other non-leaf pushing based multibit tries. Note that there remain some null entries in SRAM. In **Fig. 3 (b)**, there are four null entries, *A* ~ *D* in the SRAM. In our scheme, the root prefix of a subtrie is not expanded because the corresponding *NHI* will be stored in TCAM entry. Accordingly, the prefix *k* is not expanded. When it accesses the entry A or B, the final result is in TCAM. In case of the entry *C*, it cannot be accessed because the longer one (S4) will be matched instead. In case of the entry *D*, the final result should be the *NHI* of prefix *a*. To obtain that result the prefix *a* should have been expaned to the entry *D* by leaf-pushing. Such expansion incurs a large amount of updates when the original prefix is updated. In next section, we will describe how to find the correct matching result while avoiding leaf-pushing and large updating overhead.



subt_idx	leng	ptr	stride	NHI
*	0	S1	2	n9
11*	2	S3	1	-
0100*	4	S2	1	n1
01001*	5	S4	2	-

(a) An Indexed Multibit Trie (b) Lookup based on TCAM and SRAM

Fig. 3. Overall lookup scheme

4.2 Organization

We propose an architecture to support the *Indexed Multibit Trie* scheme in which each subtrie of a multibit trie is indexed and accessed using that index. In that architecture, TCAM contains an index to subtries and the subtrie itself is contained in SRAM. For more efficient use of the multibit trie, we split the TCAM into two parts: *p*TCAM and *n*TCAM as shown in **Fig. 4**. All subtries are divided into two groups. In one group, every subtrie root is a prefix whereas in the other group all the subtrie roots are non-prefix nodes. *p*TCAM and *n*TCAM accommodate the indexes to the former group and the latter group, respectively. For a given input IP address, each TCAM part searches for the longest matching result separately in parallel.

In **Fig. 4**, the selection logic chooses the index result (③) among the both results of TCAM parts (① and ②). It is straightforward to design the selection logic. It determines the result by means of the length of the subtrie roots. The longer (more specific) subtrie root is selected. If *leng* of ② is greater than that of ①, the selection logic chooses ②. In that case, *NHI* field of ③ is filled with that of ①. The SRAM-based search engine finds an entry with *leng*, *ptr*, *stride* and a given IP address, and gets the result of *NHI* from that SRAM entry. However, if the accessed entry is null, the final result becomes the *NHI* which comes from ③.

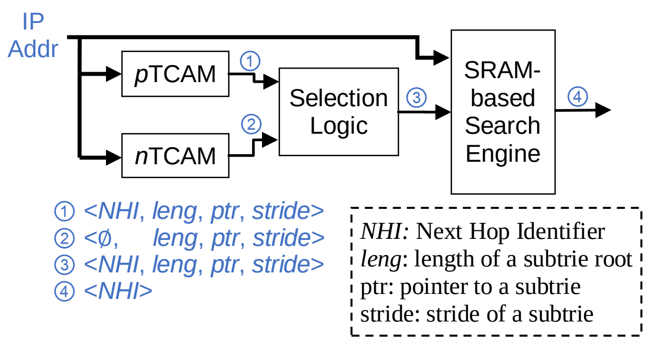
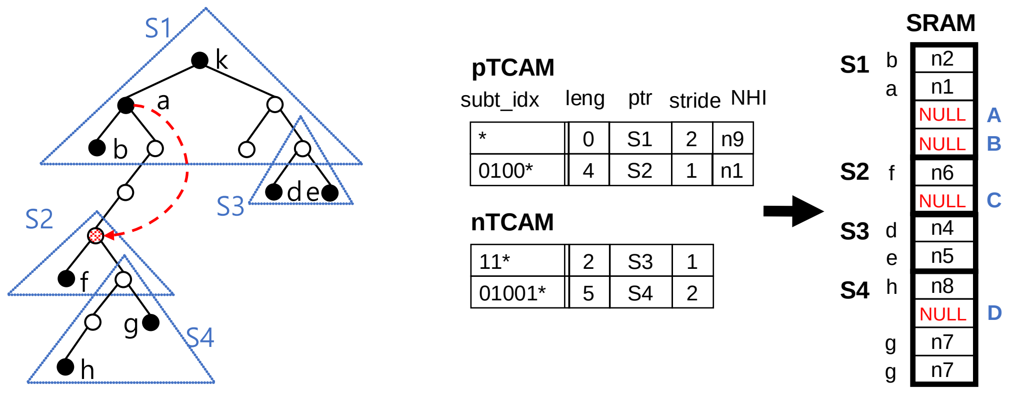

IP Addr
pTCAM
nTCAM
Selection Logic
SRAM-based Search Engine
①
②
③
④
① <NHI, leng, ptr, stride>
② <∅, leng, ptr, stride>
③ <NHI, leng, ptr, stride>
④ <NHI>
NHI: Next Hop Identifier
leng: length of a subtrie root
ptr: pointer to a subtrie
stride: stride of a subtrie

Fig. 4. Architecture for IMT-based IP lookup engine

Let us consider an example in **Fig. 5**. We do not use leaf-pushing in our scheme. When it accesses the entry *D* of SRAM in **Fig. 3**, there is a null entry even if its result should be the nexthop for prefix *a*. To resolve this problem prefix expansion is partially applied to such prefix in our scheme. To avoid heavily updating problem the expansion is limited to the immediate descendant subtrie roots, which is only the root of subtrie S2 in this example. We call it *subtrie-pushing* instead of leaf-pushing because the prefix is not expanded to leaves but to subtrie roots.

Consequently, S1 and S2 have prefixes in roots while S3 and S4 do not. So the index for the former is contained in *p*TCAM, on the other hand, that for the latter is contained in *n*TCAM. Note that the prefixes in subtrie roots are not expanded in SRAM because the corresponding *NHI*s are already in *p*TCAM. Consider null entries *A* ~ *D* in SRAM. Whenever a null entry is met in SRAM, it refers to *NHI* which was delivered from *p*TCAM. Let us consider an input IP address, 0100101. The longest matching results for *p*TCAM and *n*TCAM are $< n1, 4, \text{S2}, 1 >$ and $< \phi, 5, \text{S4}, 2 >$, respectively. Since *leng* (= 5) in *n*TCAM's result is higher, $< n1, 5, \text{S4}, 2 >$ will be delivered to SRAM-based search engine. Note that *n*1 comes from *p*TCAM's result. Now, the SRAM-based search engine accesses the entry *D* in S4 using the last two bits 01 of the IP address 0100101. The entry *D* is null, however, the lookup engine can determine the final matching result because it already has *n*1 as *NHI* which comes from *p*TCAM. When the prefix *a* is updated, it only needs to access the roots of immediate descendant subtries such as S2. So subtrie-pushing technique can save a lot of memory accesses incurred by update of a prefix.



pTCAM

subt_idx	leng	ptr	stride	NHI
*	0	S1	2	n9
0100*	4	S2	1	n1

nTCAM

11*	2	S3	1
01001*	5	S4	2

SRAM

S1	b	n2	
	a	n1	
		NULL	A
		NULL	B
S2	f	n6	
		NULL	C
S3	d	n4	
	e	n5	
S4	h	n8	
		NULL	D
	g	n7	
	g	n7	

(a) An Indexed Multibit Trie (b) IMT-based lookup

Fig. 5. An example for IMT-based lookup engine

4.3 Lookup Algorithm

Fig. 6 shows the procedure the lookup engine performs whenever an IP address is given. First, it searches both TCAMs in parallel. Then, it determines which one is longer result. The SRAM is accessed using the longer result. It will return *NHI* as the final result. If the result of the SRAM is NULL, *NHI* from pTCAM will be the final result.

```
Lookup Algorithm
Lookup(ip)
{
    do_parallel {   // Search in TCAM
        <pNHI, leng1, subt_ptr1, stride1> = lookup_pTCAM(ip);
        <leng2, subt_ptr2, stride2> = lookup_nTCAM(ip);
    }
    if (leng1 > leng2)
        <leng, subt_ptr, stride> = <leng1, subt_ptr1, stride1>;
    else
        <leng, subt_ptr, stride> = <leng2, subt_ptr2, stride2>;
    NHI = lookup_SRAM(ip, <leng, subt_ptr, stride>);

    if (NHI == NULL)
        NHI = pNHI;    // final result in pTCAM
    return NHI;
}
```

Fig. 6. Lookup algorithm

4.4 Construction of an Indexed Multibit Trie

Since a multibit trie is constituted by a set of subtries, we first obtain the subtries from a binary trie to construct an IMT. When constructing subtries, we use some metrics to ensure that each constructed subtrie is memory-efficient and fast updatable.

- **Memory efficiency of a subtrie**

The SRAM efficiency of a subtrie can be measured by

$$E_{SRAM} = n_{pref}/n_{ent} = n_{pref}/2^{stride} \tag{1}$$

where n_{pref} is the number of prefixes and n_{ent} is the number of expanded entries in the subtrie. When E_{SRAM} is calculated, n_{pref} excludes the root prefix of a subtrie because it is not actually stored in SRAM.

If the subtrie is overlapped by other descendant subtries, the SRAM efficiency will be

$$E_{SRAM} = n_{pref}/\left(2^{stride} - \Delta_{\mathrm{ent}}\right) \tag{2}$$

where Δ_{ent} is the total number of eclipsed entries in the subtrie. We allow subtries to be overlapped, which makes it possible to construct larger subtries efficiently. For example, S1 and S3 are overlapped, and Δ_{ent} is 1 in **Fig. 5**. Also, S2 and S4 are overlapped, and Δ_{ent} is 1.

The TCAM efficiency of a subtrie is,

$$E_{TCAM} = 1 - 1/n_{pref} \tag{3}$$

Both SRAM and TCAM efficiency increase as n_{pref} increases. However, E_{SRAM} tends to decrease as stride increases. E_{TCAM} is defined by 0 when n_{pref} is 1 because there is no subtrie not having any prefix. In **Fig. 5**, E_{SRAM} of S1 is 2/3 = 0.67. Likewise, those of S2, S3, and S4 are 1.0, 1.0, and 0.5, respectively. E_{TCAM} of S1, S2, S3, and S4 are 0.67, 0, 0.5, and 0.5, respectively.

- **Update overhead of a subtrie**

The average update overhead of SRAM for a subtrie can be measured by

$$U_{SRAM} = t_{aff}/n_{bt-node} = t_{aff}/\left(2^{stride+1} - 1\right) \tag{4}$$

where t_{aff} is the total number of affected entries by updates on each binary-trie node in the subtrie and $n_{bt-node}$ is the number of expanded binary-trie nodes in the subtrie except the root. In **Fig. 5**, assuming binary-trie nodes are expanded to all possible position in the subtrie, S4 has 7 (expanded) binary-trie nodes. So $n_{bt-node}$ of S4 is 6 excluding the root. t_{aff} is the sum of affected entries by means of update on each binary-trie node. t_{aff} can be computed by

$$t_{aff} = \sum A(b_i) \tag{5}$$

where b_i is the *i*-th binary-trie node in the subtrie except the root and $A(x)$ denotes the number of entries affected by binary-trie node x. For example, in S4, update on prefix g affects 2 entries in the subtrie, so $A(g)$ = 2. When all $A(b_i)$ are computed similarly and summed up, t_{aff} of S4 is 7.

If the subtrie is overlapped by other descendant subtries, the average update overhead will be

$$U_{SRAM} = t_{aff}/\left(2^{stride+1} - 1 - \Delta_{\mathrm{bt-node}}\right) \tag{6}$$

where $\Delta_{\mathrm{bt-node}}$ is the total number of eclipsed binary-trie nodes in the subtrie. When calculating t_{aff}, it excludes updates on eclipsed nodes. Since all binary-trie nodes except the root affects only SRAM, we do not consider the average update overhead of TCAM for a subtrie.

For a given subtrie, maximum number of affected entries by update is

$$m_{aff} = \mathrm{Max}(A(b_1), A(b_2), \ldots, A(b_n)) \tag{7}$$

When constructing each subtrie, we consider E_{SRAM} but not U_{SRAM} because U_{SRAM} does not increase as E_{SRAM} increases in general. Also, m_{aff} is considered to control the excessive accesses to SRAM. **Fig. 7** shows our IMT construction algorithm. The algorithm uses two parameters α and β to decide whether it makes the current subtrie to be enlarged or not. α is the lower bound of E_{SRAM}, i.e., every constructed subtrie will have the value at least α. β is the upper bound of m_{aff}, i.e., every constructed subtrie will have the value at most β.

In the IMT construction algorithm, among all binary-trie nodes only prefix nodes are visited in reverse-level order, i.e., from bottom to top order. The function **next_prefix_node**() gives such nodes in turn. The currently visited prefix node *p* directly constitutes a subtrie. Then, it repeats to check if the current subtrie can be enlarged to cover the sibling node with stride incremented by one. If the enlarged subtrie does not satisfy $\alpha \leq E_{SRAM}$ and $\beta \geq m_{aff}$, the previous subtrie is established. Whenever a subtrie is constituted, it just marks the root node in the binary trie and sets its stride instead of actual prefix expansion. The prefix expansion is straightforward and will be done at later phase. Every prefix is reviewed and contained just once in a subtrie by dynamic programming.

Indexed Multibit Trie Construction Algorithm

```
Construct_IMT(α, β)
{
    while (there remains any prefix node) {
        p = next_prefix_node( );
        if (p.flag == true) continue;
        stride = 1;
        pref = 1;     // prefixes of the current subtrie
        ent = 0;// eclipsed entries of the current subtrie
        while (p.leng > 0) {
            q = p’s sibling node;
            tpref = get_pref(q, stride) + pref;
            tent = get_ent(q, stride) + ent;
            tmax_aff = MAX(get_max_aff(q, stride), max_aff);
            if (mem_eff(tpref, tent, stride) < α || tmax_aff > β)
                break;
            pref = tpref;
            ent = tent;
            max_aff = tmax_aff;
            p = p’s parent;
            if (p is a prefix) prefixes ++;
            stride++;
        }
        p.stride = stride – 1;
        p.subt_root = true;     // Set a new subtrie root
        Mark all nodes’ flag as true in the subtrie;
    }
}
```

Fig. 7. IMT construction algorithm

In **Fig. 5**, we have shown prefix *a* is expanded to the roots of the immediate descendant subtries by *subtrie-pushing*. For a given binary-trie node *b*, the number of immediate descendant subtrie roots of *b* is denoted by $\delta(b)$ when there is no prefix node on the path from *b* to the immediate descendant subtrie roots including those roots. If *b* is the root of a subtrie, $\delta(b) = 0$. Suppose $\delta(a)$ is large in **Fig. 5**. It implies the update on the prefix *a* causes a lot of accesses to TCAM. In that case, we need to control excessive subtrie-pushing by creating a new subtrie. Note that we don’t have to do subtrie-pushing if the prefix *a* becomes the root of a new subtrie. **Fig. 8** shows such splitting algorithm. γ is a parameter to limit the number of immediate descendant subtries. For some prefix *p*, if $\delta(p)$ is greater than γ, the new subtrie will be split with *p* as the root.

Subtrie Splitting Algorithm

```
Split_Subtries(p, cur_stride, $\gamma$)
{
    if (p.subt_root)
        cur_stride = p.stride;
    if ( !p.subt_root && p is a prefix && $\delta(p) > \gamma$) {
        // Set a new subtrie root
        p.subt_root = true;
        p.stride = cur_stride;
    }
    if (cur_stride > 0) {
        Split_Subtries(p.lchild, cur_stride - 1);
        Split_Subtries(p.rchild, cur_stride - 1);
    } else {
        Split_Subtries(p.lchild, 0);
        Split_Subtries(p.rchild, 0);
    }
}
```

Fig. 8. Subtrie splitting algorithm

4.5 Update Algorithm

Each update message on a prefix is associated with a binary-trie node. There are three cases in position of that node: at subtrie root, inside subtrie, and outside subtrie. Fig. 9 depicts the algorithm according to the updating node position. First, if the updating node is a subtrie root, only TCAM is accessed for the update. It incurs at most one memory access to *p*TCAM and *n*TCAM each except movements to preserve order of prefixes in TCAM. Some technique to reduce memory movements for preserving order in TCAM has been researched [12]. The TCAM entries in our scheme are much less than others, so we expect the memory movement for preserving order is not crucial in our scheme. Second, if a new prefix is inserted outside

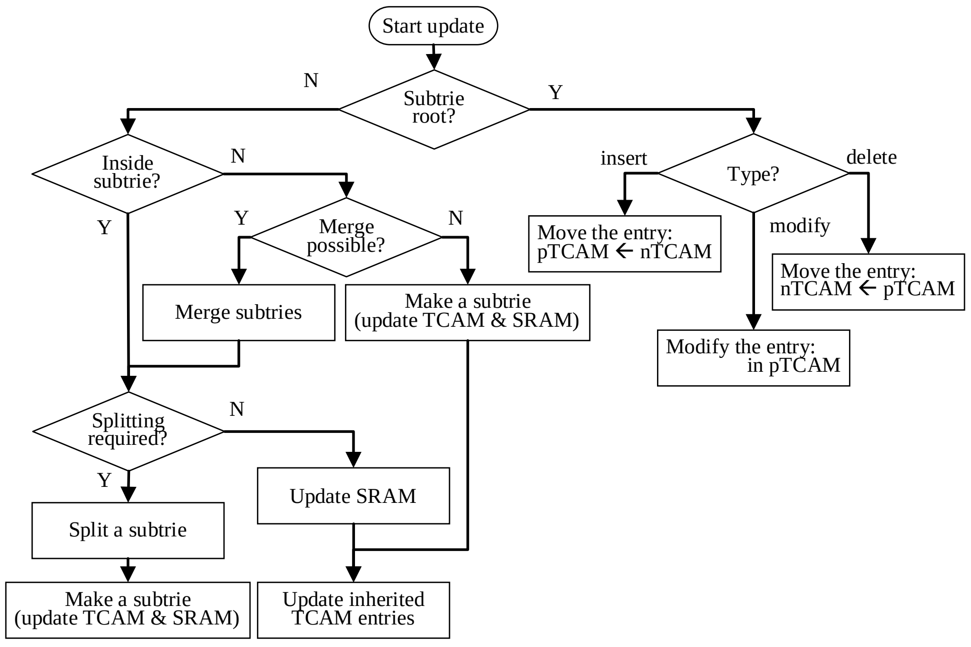

Start update
Subtrie root?
N
Y
Inside subtrie?
N
Y
Merge possible?
Y
N
Merge subtries
Make a subtrie (update TCAM & SRAM)
Splitting required?
N
Y
Split a subtrie
Update SRAM
Make a subtrie (update TCAM & SRAM)
Update inherited TCAM entries
Type?
insert
delete
modify
Move the entry: pTCAM ← nTCAM
Move the entry: nTCAM ← pTCAM
Modify the entry: in pTCAM

Fig. 9. Update Algorithm

subtrie, it should be checked whether the new prefix can be merged with the existing subtrie. The criterion for merging is whether U_{SRAM} is larger than β after merging. If not, a new subtrie will be created and *p*TCAM is accessed once. Then, several subtrie-pushed entries in *p*TCAM should be updated. Lastly, in case the updating node is inside some subtrie, it is checked whether the subtrie can be split or not. U_{SRAM} is used for criterion. In all the cases the number of SRAM and TCAM accesses are limited by β and γ, respectively.

5. Evaluation

In this section, the proposed architecture is evaluated in terms of memory efficiency and updatability using real-world public routing tables. For the experiment, the routing tables were collected on three different dates from ripe.net [22]. Since the network prefixes do not vary much in different locations, we used routing data from various dates to observe changes over time. **Table 1** depicts the characteristics of the routing tables. All the routing tables were converted into IPv4 Forwarding Information Bases (FIBs) for our experiment. The experiment was performed using Intel(R) Core(TM) i7-2600 (3.4 GHz) with main memory of 4 GB.

Table 1. Routing tables

	rrc0-2017	rrc0-2015	rrc0-2013
Location	Amsterdam	Amsterdam	Amsterdam
Date	03/01/2017	03/01/2015	03/01/2013
Prefixes	668,390	550,463	484,860

Table 2 shows the summary of IMT construction results with parameters (α = 0.5, β = 64, γ = 16). In the table, the number of prefixes has been increased sharply over time, so the memory efficiency is crucial in IP lookup engine to accommodate more prefixes. Note that the other characteristics in the table have a tendency to be time-invariant except the requirement of *p*TCAM, *n*TCAM, and SRAM. For example, the average length of the subtrie roots is about 21, which is irrespective of collecting date.

Table 2. IMT construction results (α = 0.5, β = 64, γ = 16)

	rrc0-2017	rrc0-2015	rrc0-2013
Prefixes	668,390	550,463	484,860
Binary-trie nodes	1,533,248	1,292,006	1,154,517
Subtries	123,097	112,011	102,809
Avg. leng of subtrie root	21.01	21.02	21.05
Max. leng of subtrie root	31	31	31
Avg. stride of subtrie	2.15	2.12	2.09
Max. stride of subtrie	11	11	11
Avg. multibit-trie depth	5.19	5.20	5.09
Max. multibit-trie depth	12	12	12
pTCAM(entries)	50,446	45,299	41,361
nTCAM(entries)	72,651	66,712	61,448
SRAM(entries)	1,114,798	931,458	825,112
Eclipsed(entries)	155,820	132,554	119,434
TCAM power(nJ)	97.8	80.8	71.3
TCAM search delay(ns)	7.1	5.4	4.6

In **Table 2**, ‘multibit-trie depth’ means the highest level of the trie, i.e., maximum number of subtries on the path from the root to each leaf node. It is related to the number of memory accesses when a conventional SRAM-based multibit-trie is used. Its average value is about 5 and the maximum value is 12. On the other hand, in IMT, it takes at most one TCAM access and one SRAM access.

Moreover, IMT requires less memory compared to the conventional SRAM-based multibit-trie. In rrc0-2017, about 123 K entries are required for TCAM and 1.1 M entries are required for SRAM. Since each SRAM entry stores only *NHI* field, it requires at most 16 bits. As a result, the total SRAM requirement is 2.2 MB in IMT. Since the conventional SRAM-based multibit-trie does not use TCAM, it only requires SRAM of 2.2 M entries. However, in that case, each SRAM entry needs a pointer to the child subtrie, so each entry requires 32 bits additionally and total SRAM requirement becomes 6.6 MB. In **Table 2**, ‘eclipsed’ is the size of an overlapped region by descendant subtries which is explained in Section 4.4. The eclipsed space can be reused like free space, so we can reduce the total SRAM requirement by the amount of the eclipsed entries. As a result, the required SRAM is reduced by 14 %.

TCAM plays an important role in parallel search of subtries, but it may cause much power to be consumed. Agrawal and Sherwood presented TCAM power consumption and delay time model useful for network system design [23]. We obtained TCAM power and search delay of IMT using their model and tool. For calculation of power and delay, TCAM size and technology feature size are used as input parameters. For 90nm technology, the results of TCAM power and delay are shown in **Table 2**.

In Section 4.4, m_{aff} denotes the maximum number of entries affected by a single node update. Actually, it represents the maximum number of SRAM accesses incurred by a single update. **Fig. 10** shows the distribution of m_{aff}, in which most subtrie updates (99%) incur less than 17 SRAM accesses. The largest value of m_{aff} is 1276 when we set the parameters as α = 0.5, $\beta = \infty$, $\gamma = \infty$. In case of α = 0.5, β = 64, γ = 16, every subtrie is constructed to have the value of m_{aff} less than or equal to 64, however, the percentage of m_{aff} being 17 ~ 64 is less than 1% as stated above. In addition, the actual SRAM access time can be reduced because the contiguous entries can be accessed in burst mode and each entry is merely 16-bit long. In **Fig. 10**, some count values are higher than others, especially when the number of SRAM accesses is a power of two. The reason for this is that the number of entries in a subtrie is always a power of two and the maximum number of entries affected by a single update is most often a power of two.

In the earlier section, $\delta(b)$ denotes the number of immediate descendant subtrie roots of a binary-trie node b when there is no prefix node on the path from b to the subtrie roots including those roots. It presents how many subtrie roots are affected when a binary trie node is updated. Since the subtrie roots are always stored in TCAM, δ means the number TCAM accesses incurred by a binary-trie node update. **Fig. 11** shows the distribution of δ and the number of TCAM accesses incurred by a single update is lower than 4 (99%). The largest value of δ is 890, however, the value can be limited by 16 when we set the parameters as α = 0.5, β = 64, γ = 16.

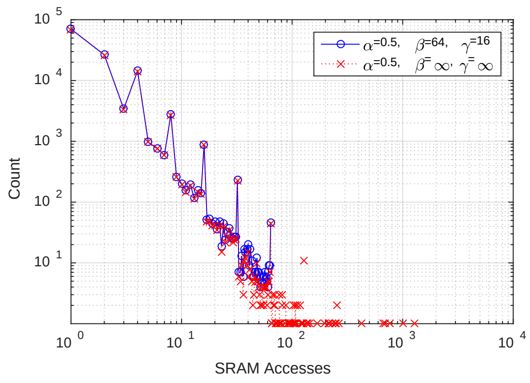


Fig. 10. Distribution of m_{aff} in rrc0-2017

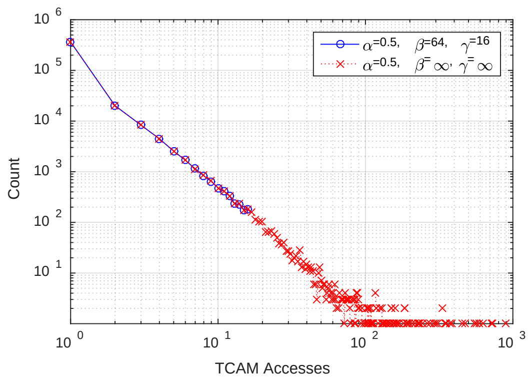


Fig. 11. Distribution of δ in rrc0-2017

Fig. 12 shows how the number of subtries and the stride size change with α. The number of subtries steadily increases with α because the size of each subtrie is reduced as α increases. On the other hand, average stride goes down as α increases. When α is higher than 0.5, the average stride becomes about 1. It implies each subtrie has merely two entries in average.

Fig. 13 shows the number of entries in TCAM and SRAM varying with α. As α grows, the required SRAM size sharply decreases while the required TCAM size gradually increases. **Fig. 14** and **Fig. 15** show the required TCAM size and SRAM size, respectively. In **Fig. 14**, the size of *p*TCAM is significantly increased compared to that of *n*TCAM when α is higher than 0.5. This is due to the increase in the number of subtrie roots contained in the *p*TCAM as the number of small subtries increases.

The memory efficiency and the update overhead are depicted in **Fig. 16** and **Fig. 17**, respectively. There are 6 graphs in each figure according to routing data collection dates and the type of memory. **Fig. 16** shows memory efficiency of TCAM and SRAM. Both memories are inversely related to each other. However, the trend of the change is independent of the routing data collection dates. **Fig. 17** shows the average number of memory accesses incurred by a single update. These values for TCAM and SRAM also change inversely each other, but they do not change over time. It implies that the characteristics of the IMT, such as memory efficiency and update overhead, hardly change with time.

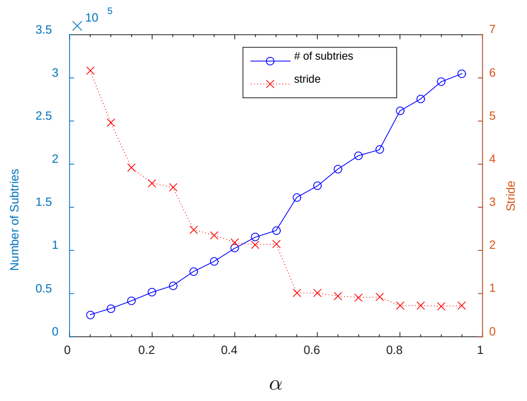


Fig. 12. Subtries and stride in rrc0-2017

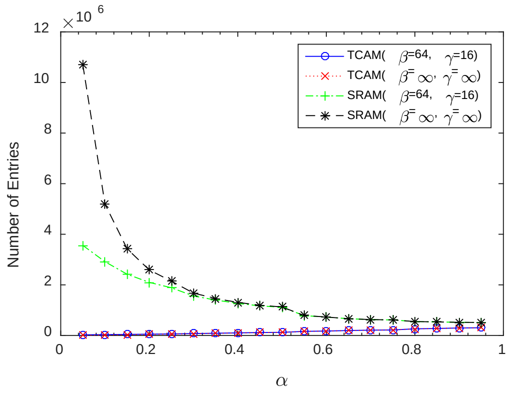


Fig. 13. Memory requirements in rrc0-2017

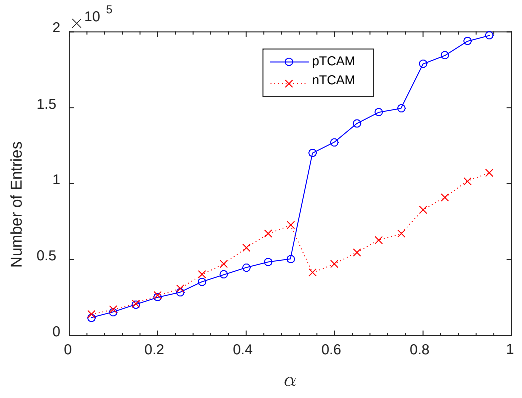


Fig. 14. TCAM requirement in rrc0-2017

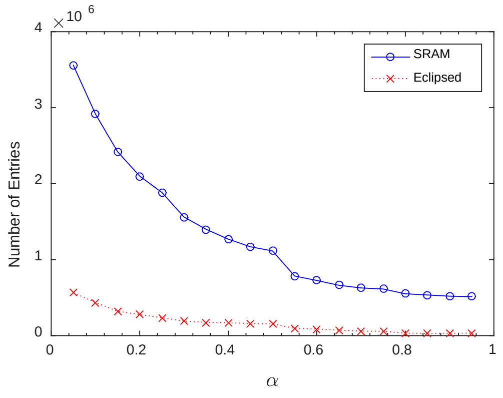


Fig. 15. SRAM requirement in rrc0-2017

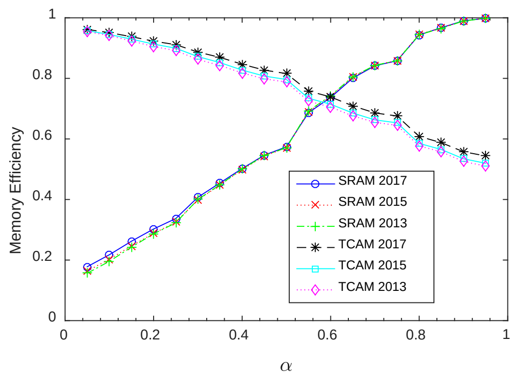


Fig. 16. Memory efficiency

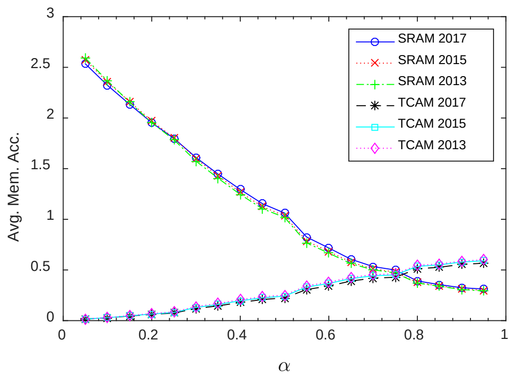


Fig. 17. Update overheads

IMT is compared to several architectures with respect to the number of accesses to TCAM and SRAM, and also their required size, in **Table 3**. ‘Uniform TCAM’ denotes the architecture which simply consists of a single TCAM. ‘CoolCAM-subtree’ and ‘CoolCAM-postorder’ are the schemes described in [7]. ‘1-12Wc’ and ‘M-12Wb’ are the best-effort schemes in [8]. Our IMT is constructed with $\alpha = 0.5$, $\beta = 64$, $\gamma = 16$. For the compared schemes, we used a bucket size of 128 entries because power consumption in those schemes becomes the smallest when the bucket size is 128 [8]. In the table, the size of TCAM and SRAM is measured by the number of entries. In IMT, the size of SRAM is reduced by a factor of 144 bits since 144-bit wide SRAM was assumed in [8]. Our scheme shows the smallest memory requirement as well as fewer TCAM searches.

Table 3. Comparison on memory size and accesses in rrc0-2017

Scheme	TCAM Size	SRAM Size	TCAM Searches	SRAM Accesses
Uniform TCAM	668,390	668,390	1	1
CoolCAM-subtree	1,347,224	1,357,668	2	2
CoolCAM-postorder	741,496	741,496	2	2
1-12Wc	267,774	267,774	2	2
M-12Wb	142,451	142,451	2	2
IMT	123,097	121,014	1	2

Overall memory efficiency can be evaluated by considering the relative cost of TCAM and SRAM. **Fig. 18** depicts the overall memory efficiency in each scheme under the general fact that the numbers of transistors per cell of TCAM and SRAM are 16 and 6, respectively. IMT gives better overall memory efficiency than other schemes while $\alpha \leq 0.5$. The overall memory efficiency can be also controlled by α in our scheme while the other schemes have little change with their bucket size.

The proposed scheme is designed not only to focus on memory efficiency and update overhead, but also has the advantage of being controlled by three parameters, α, β, and γ. The above experiment results are summarized and discussed in terms of memory efficiency and update overhead as follows.

First, the TCAM requirement is much lower than the SRAM requirement, though the memory requirements of TCAM and SRAM are inversely related to each other. Considering that the number of transistors per cell and the power consumption are high in the TCAM, it is desirable to make the TCAM size as small as possible. In addition, the TCAM requirement sharply increases when α is larger than 0.5, while the SRAM requirement is generally low when α is 0.4 or more. Therefore, it is thought that an optimal memory requirement can be obtained when α is around 0.5. The memory efficiency of TCAM and SRAM is also inversely related to each other. Considering the number of transistors per cell of TCAM and SRAM, the overall memory efficiency of IMT gradually decreases with α. However, if α is 0.5 or less, it always gives better results than other schemes.

Second, the update overhead also shows an inverse relationship between TCAM and SRAM, as does the memory efficiency. With α = 0.5, for a single update, the average numbers of SRAM accesses and TCAM accesses are 1.06 and 0.22, respectively. In other words, the number of SRAM memory accesses is higher than that of TCAM. However, the effective update overhead of SRAM is expected to be relatively low because SRAM latency is much lower than TCAM latency and it can operate in burst mode.

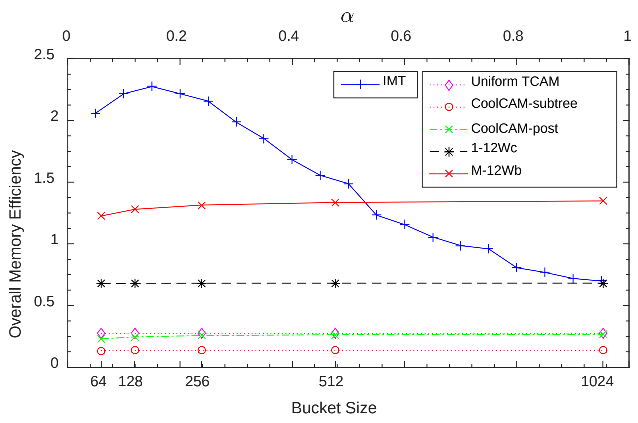


Fig. 18. Comparison on overall memory efficiency in rrc0-2017

6. Conclusion

TCAM-based IP address lookup engine can find the longest matching prefix with one access using parallel search, however, its power consumption and cost have been the problems. Even the current FIB size is too large to be contained in a single TCAM. On the other hand trie-based IP address lookup uses SRAM, which is cheaper and also consumes less power than

TCAM. However, the trie-based IP address lookup usually has to traverse many nodes, which causes the lookup performance to be degraded. Though multibit trie-based approach can reduce the traversing steps significantly, it still needs to access the SRAM several times. Also, an update incurs access to several SRAM entries and sometimes a large number of entries.

In this paper, we propose a novel multibit trie scheme, *Indexed Multibit Trie* (*IMT*) and an architecture based on it. In the IMT, each subtrie is indexed and accessed directly without going through intermediate subtries. We use TCAM to store such index because only the longest matching index can be used to access the target subtrie. In the proposed architecture, IP address lookup is performed very fast requiring maximum two memory accesses. One access is for a subtrie index and the other is for a subtrie entry regardless of the depth in IMT.

Subtrie partitioning is crucial to save memory and enable fast updatability. Generally, the larger subtrie increases the requirement of SRAM space but decreases the requirement of TCAM space. In this paper, we use three criteria α, β, and γ to construct the IMT efficiently. The size of SRAM and TCAM can be well-controlled using those parameters. Also, using those parameters the update overhead is controlled not to excessively access the memories. Experiment results with real-world FIBs show that the proposed scheme can achieve good memory efficiency as well as fast updatability by setting appropriate parameters.

References

[1] V. Fuller, T. Li, J. Yu, and K. Varadhan, “Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy,” *RFC1519*, 1993. Article (CrossRef Link).

[2] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, “Survey and Taxonomy of IP Address Lookup Algorithms,” *IEEE Network*, vol. 15, issue 2, pp. 8-23, March/April 2001. Article (CrossRef Link).

[3] V. Srinivasan and G. Varghese, “Fast Address Lookups Using Controlled Prefix Expansion,” *ACM Transactions on Computer Systems*, vol. 17, no. 1, pp. 1–40, February 1999. Article (CrossRef Link).

[4] H. Liu, “Routing Table Compaction in Ternary CAM,” *IEEE Micro*, vol. 22, issue 1, pp. 58-64, Jan./Feb. 2002. Article (CrossRef Link).

[5] L. Luo, G. Xie, Y. Xie, L. Mathy, and K. Salamatian, “A Hybrid Hardware Architecture for High-Speed IP Lookups and Fast Route Updates,” *IEEE/ACM Transactions on Networking*, vol. 22, no. 3, June 2014. Article (CrossRef Link).

[6] J. Kim, M.-C.l Ko, H.-K. Kang, and J. Kim, “A Hybrid IP Forwarding Engine with High Performance and Low Power,” in *Proc. of International Conference on Computational Science and Its Applications*, pp. 888-899, 2009. Article (CrossRef Link).

[7] F. Zane, G. Narlikar, and A. Basu, “CoolCAMs: Power-Efficient TCAMs for Forwarding Engines,” in *Proc. of IEEE INFOCOM*, vol. 1, pp. 42–52, 2003. Article (CrossRef Link).

[8] W. Lu and S. Sahni, “Low-Power TCAMs for Very Large Forwarding Tables,” *IEEE/ACM Transactions on Networking*, vol. 18, no. 3, June 2010. Article (CrossRef Link).

[9] G. Wang and N.-F Tzeng, “Exact Forwarding Table Partitioning for Efficient TCAM Power Savings,” in *Proc. of IEEE NCA 2007*, pp. 249-252, 2007. Article (CrossRef Link).

[10] M. J. Akhbarizadeh, M. Nourani, and C. D. Cantrell, “Prefix Segregation Scheme for a TCAM-Based IP Forwarding Engine,” *IEEE Micro*, vol. 25, issue 4, pp. 48-63, July-August 2005. Article (CrossRef Link).

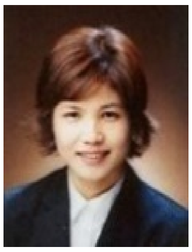
[11] M. J. Akhbarizadeh, M. Nourani, R. Panigrahy, and S. Sharma, “A TCAM-Based Parallel Architecture for High-Speed Packet Forwarding,” *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 58-72, January 2007. Article (CrossRef Link).

[12] D. Shah and P. Gupta, “Fast Updating Algorithms for TCAMs,” *IEEE Micro*, vol. 21, no. 1, pp. 36-47, Jan.-Feb. 2001. Article (CrossRef Link).

[13] H. Le, W. Jiang, and V. K. Prasanna, “A SRAM-based Architecture for Trie-based IP Lookup using FPGA,” in *Proc. of 16th IEEE International Symposium on Field-Programmable Custom Computing Machines*, pp. 33–42, 2008. Article (CrossRef Link).
[14] Anindya Basu and Girija Narlikar, “Fast Incremental Updates for Pipelined Forwarding Engines,” *IEEE/ACM Transactions on Networking*, vol. 13, no. 3, pp. 690-703, June 2005. Article (CrossRef Link).
[15] J. Lee and H. Lim. "Multi-Stride Decision Trie for IP Address Lookup," *IEIE Transactions on Smart Processing & Computing*, vol. 5, no. 5, pp.331-336, 2016. Article (CrossRef Link).
[16] Y. Wu, G. Nong, and M. Hamdi, "Scalable Pipelined IP lookup with Prefix Tries," *Computer Networks*, vol 120, pp. 1-11, June 2017. Article (CrossRef Link).
[17] Hung-Mao Chu, Tsung-Hsien Li, and Pi-Chung Wang, “IP Address Lookup by Using GPU,” *IEEE Transactions on Emerging Topics in Computing*, vol. 4, issue 2, April-June 2016. Article (CrossRef Link).
[18] Yanbiao Li, Dafang Zhang, Alex X. Liu, and Jintao Zheng, “GAMT: A Fast and Scalable IP Lookup Engine for GPU-based Software Routers,” in *Proc.of 9th ACM/IEEE ANCS’13*, pp. 1-12, 2013. Article (CrossRef Link).
[19] Sartaj Sahni and Kun Suk Kim, “Efficient Construction of Multibit Tries for IP Lookup,” *IEEE/ACM Trans. on Networking (TON)*, vol. 11, issue 4, pp. 650–662, August 2003. Article (CrossRef Link).
[20] Stefan Nilsson and Gunnar Karlsson, “IP-Address Lookup Using LC-Tries,” *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083-1092, June 1999. Article (CrossRef Link).
[21] Yanbiao Li, Dafang Zhang, Kun Huang, Dacheng He, and Weiping Long, “A Memory-Efficient Parallel Routing Lookup Model with Fast Updates,” *Computer Communications*, vol. 38, pp. 60-71, 2014. Article (CrossRef Link).
[22] RIS Raw Data. Article (CrossRef Link).
[23] B. Agrawal and T. Sherwood, “Ternary CAM Power and Delay Model: Extensions and Uses,” *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 16, issue 5, pp. 554–564, 2008. Article (CrossRef Link).

Junghwan Kim received the B.S., M.S. and Ph.D degrees from Seoul National University, Seoul, in 1991, 1993 and 1999, respectively, all in computer science. In 1999 he joined Samsung Electronics, where he was a senior researcher. In 2001 he joined the faculty of Konkuk University, where he is now a professor. His research interests are in the areas of parallel computing, communication networking, GPU computing, and design of efficient algorithms.

Myeong-Cheol Ko is a professor of computer engineering at Konkuk University, where he directs the AVIT(Advanced Visualization and Interaction Technology) research group. He received PhD in computer science from Yonsei University in 2003. His research interests are in 3D computer graphics and human-computer interaction focusing on the design and implementation of augmented reality systems.



Moonsun Shin received Ph.D degrees from Chungbuk National University in 2004 in computer science. In 2005 she joined the faculty of Konkuk University, where she is now an associate professor. Her research interests include context awareness, security model. ICT convergence and big data analysis.

Jinsoo Kim received the B.S. degree from Seoul National University, Seoul, in 1983, and the M.S. and Ph.D degrees from Korea Advanced Institute of Science and Technology (KAIST), in 1985 and 1998, respectively, all in computer engineering. In 1985, he joined Korea Telecom, where he was a senior researcher. In 2000, he joined the faculty of Konkuk University, where he is now a professor. His research interests include parallel computing architectures, high-speed networking, packet processing systems, and named data networking.