# Dynamic Tree Bitmap For IP Lookup and Update [*]

Sartaj Sahni  Haibin Lu
{sahni, halu}@cise.ufl.edu
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

**Abstract**

We propose a data structure–dynamic tree bitmap–for the representation of dynamic IP router tables that must support very high lookup and update rates. In fact, the dynamic tree bitmap is able to support updates at the same rate as lookups and is very competitive with other structures–tree bitmap and BaRT–proposed earlier for dynamic tables. Although the dynamic tree bitmap requires more memory than is required by the tree bitmap and BaRT structures, the required memory remains reasonable. The real value of our structure is its ability to support a very high update rate.

**Keywords**: Packet forwarding, dynamic router tables, longest-prefix matching.

## 1 Introduction

An Internet router classifies incoming packets into flows[1] utilizing information contained in packet headers and a table of (classification) rules. This table is called the *rule table* (equivalently, *router table*). In this paper, we assume that packet classification is done using only the destination address of a packet. Each rule-table rule is a pair of the form $(F, NH)$, where $F$ is a filter and $NH$ is a next hop. In this paper, we assume that each filter is a destination-address prefix and that a filter *matches* all destination addresses for which it is a prefix. For example, the filter 10* matches all destination addresses that begin with the bit sequence 10; the length of this prefix is 2. Since an Internet rule-table may contain several rules that match a given destination address $d$, a tie breaker is used to select a rule from the set of rules that match $d$. Traditionally, ties are broken by selecting the next hop associated with the longest prefix that matches the packet's destination address. In this paper, we focus on longest-prefix matching. We refer to the associated rule tables as LMPTs. We use $W$ to denote the maximum possible length of a prefix. In IPv4, $W = 32$ and in IPv6, $W = 128$.

Data structures for longest-prefix matching have been intensely researched in recent years. Ternary content-addressible memories, TCAMs, use parallelism to achieve $O(1)$ lookup [22]. Each memory cell of

[*]This research was supported, in part, by the National Science Foundation under grant ITR-0326155

[1]A *flow* is a set of packets that are to be treated similarly for routing purposes.

a TCAM may be set to one of three states 0, 1, and don't care. The prefixes of a router table are stored in a TCAM in descending order of prefix length. Assume that each word of the TCAM has 32 cells. The prefix 10* is stored in a TCAM word as 10???...?, where ? denotes a don't care and there are 30 ?s in the given sequence. To do a longest-prefix match, the destination address is matched, in parallel, against every TCAM entry and the first (i.e., longest) matching entry reported by the TCAM arbitration logic. So, using a TCAM and a sorted-by-length linear list, the longest matching-prefix can be determined in $O(1)$ time. A prefix may be inserted or deleted in $O(q)$ time, where $q$ is the number of different prefix lengths in the table [31]. Although TCAMs provide a simple and efficient solution for static and dynamic router tables, this solution requires special hardware, costs more, and uses more power and board space than solutions that employ SDRAMs. EZchip Technologies, for example, claim that classifiers can forgo TCAMs in favor of commodity memory solutions [2, 21]. Algorithmic approaches that have lower power consumption and are conservative on board space at the price of slightly increased search latency are sought. "System vendors are willing to accept some latency in their searches if it means lowering the power of a line card" [21].

Ruiz-Sanchez, Biersack, and Dabbous [24] review data structures for static LMPTs and Sahni, Kim, and Lu [30] review data structures for both static and dynamic LMPTs. Although several data structures have been proposed for dynamic LMPTs [34, 27, 28, 14, 15, 16, 17], these structures improve insert/delete complexity at the expense of lookup complexity. Eatherton et al. [1] have have proposed the tree bitmap (TBM) data structure for dynamic LMPTs. Although this data structure results in fast lookups, inserts and deletes make an excessive number of memory accesses. The reasons for the poor performance of TBM on insert and delete operations stem from the fact that, in a TBM, the children of a node are stored in contiguous memory using variable-size memory blocks. This has two detrimental consequences for insert and delete operations:

1. A complex memory management system is needed to allocate and deallocate variable size memory blocks. In the specific design detailed in [1], 17 different block sizes are employed. The memory management system described by them requires over 3000 memory accesses [2], in the worst-case, to allocate a node.

[2]Although [1] asserts that an allocation can be done with 1892 memory accesses, their analysis is flawed. In TBM, the memory is divided into 17 parts: 2-node blocks, 3-node blocks, 4-node blocks, $\cdots$, and 18-node blocks. The worst case for memory allocation occurs when an 18-node block is to be allocated and the only free space available is between the 2-node memory and the top of memory. To free space for an 18-node block, one needs to move 2 17-node blocks up; to free space for 2 17-node blocks, one needs to move 3 16-node blocks up; $\cdots$; and to free space for 36 3-node blocks, one needs to move 54 2-node blocks up. Hence, the amount of memory that needs to be passed, in the worst case, from end to end is not 34 nodes as claimed in [1].

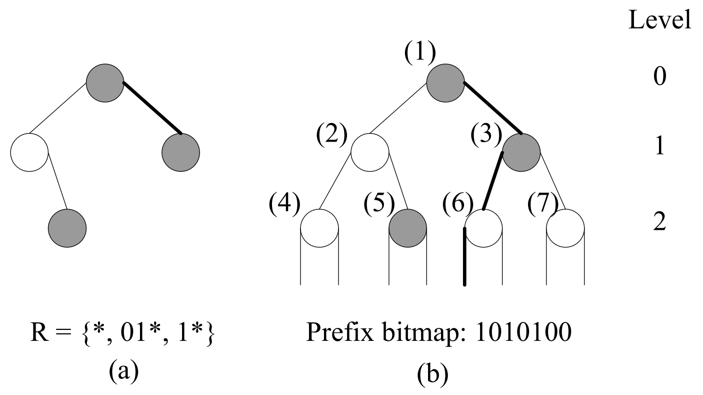

Figure 1: (a). One bit trie. (b). Corresponding full trie with height 2. The shaded nodes store next hops for the associated prefixes

2. The insertion/deletion of a prefix may change the number of children that a node has. This requires the deallocation of an old memory block and allocation of a new memory block. Consequently, insertion and deletion require more than 3000 memory accesses each, in the worst case.

In this paper, we propose an alternative tree bitmap scheme, which we call dynamic tree bitmap (DTBM). In DTBM, each node has an array of child pointers. This change greatly simplifies memory management and results in much faster inserts and deletes; lookup speed is not sacrificed. Although the DTBM uses more memory than does a TBM, the required memory is well within the capacity of commercially available SRAMs. We propose two alternatives–packed and lazy–to the basic (called free-mode) DTBM data structure. The lazy-mode alternative is expected to have very good performance in real-world applications. The DTBM structure is described in Section 2 and experimental results presented in Section 3.

## 2 Dynamic Tree Bitmap (DTBM)

### 2.1 The Data Structure

Figure 1 (a) shows the one-bit trie representation of the prefix set $R = \{*, 01*, 1*\}$. For any node $x$ in the one-bit trie, let $prefix(x)$ be an encoding of the path from the root to $x$. Left branches add a 0 to the encoding and right branches add a 1. The encoding always ends in a $*$. So, $prefix(root) = *$, $prefix(root.left) = 0*$, $prefix(root.right) = 1*$, $prefix(root.left.left) = 00*$, and so on. In our figure, a node $x$ is shaded iff $prefix(x) \in R$.

Suppose we extend the one-bit trie of Figure 1(a) to a full trie whose height is 2 (Figure 1(b)). This full trie may be compactly represented using a prefix bitmap [1] (also known as the internal bitmap

(IBM)) that is obtained by placing a 1 in every shaded node and a 0 in every non-shaded node and then listing the node bits in level order. For the full trie of Figure 1(b), this level order traversal results in the IBM 1010100. Notice that the IBM of a full one-bit trie of height $h$ has exactly $2^{h+1}-1$ bits, one bit for each node in the trie. Note also that if we number the bits of the IBM in level order beginning at 1, then we can simulate a walk through the corresponding full trie by using the formulae: left child of $i$ is $2i$, right child is $2i+1$ and parent is $i/2$ [29]. To obtain the next hop for the longest matching prefix, we supplement the IBM with a next hop array, $nextHop$, that stores the next hops stored in the nodes of the full trie. For a full trie whose height is $h$, the size of the next hop array is $2^{h+1}-1$. For our example trie, the next-hop array has 7 entries, 4 of which are $null$.

One way to find the next hop associated with the longest matching prefix is to simulate a top-to-bottom walk beginning at the trie root and using the bits in the destination address. The simulated walk moves through the bits of the IBM keeping track of the last 1 that is seen. If the last 1 was in position $i$ of the IBM, we retrieve the next hop from position $i$ of the next-hop array.

An alternative strategy is to start from the appropriate leaf of the trie and move up toward the root stopping at the first 1 that is encountered in the IBM. Let $d.bits(j)$ be the integer represented by the first $j$ bits of the destination address $d$. Note that $d.bits(0) = 0$ for all $d$ and that for $d = 101_2$, $d.bits(1) = 1_2 = 1$, $d.bits(2) = 10_2 = 2$, and $d.bits(3) = 101_2 = 5$. In case $d$ has fewer than $j$ bits, then $d.bits(j)$ is obtained by first appending enough 0s to $d$ so that the new $d$ has $j$ bits. For a full trie whose height is $h$, a top-down search terminates at the leaf whose index is $2^h + d.bits(h)$. For example, when $d = 101_2$ and $h = 2$, the search terminates at the leaf labelled $2^2 + 2 = 6$ in Figure 1(b). In the alternative strategy, we start at bit $2^h + d.bits(h)$ of the IBM and simulate a walk up the trie by dividing the current bit index by 2 for each move to a parent. The simulated walk terminates when we either reach a bit whose value is 1 or when the bit index becomes 0. Figure 2 gives the algorithm that uses this alternative strategy to find the length of the longest matching prefix for $d$. $IBM(i)$ returns the $i$th bit of the IBM (note that bits are numbered beginning at 1).

Since the height of the one-bit trie for an IPv4 (IPv6) prefix set may be as large as 32 (128), it isn't practical to represent the prefix set by the IBM and next-hop array for the corresponding full one-bit trie. Instead, we pick a stride $s$ and construct the stride $s$ full extension (SFE) of the one-bit trie. The SFE is obtained by partitioning the one-bit trie into subtries whose height is $s-1$, beginning at the root; partitions that have no descendants may have a height smaller than $s-1$. Each partition is then expanded to a full trie whose height is $s-1$. Figure 3 shows an example stride-3 SFE. In partition $x_4$,

```
**Algorithm** $llmp(d)\{$
    $i = 2^h + d.bits(h);$
    $length = h;$
    **while**$(i > 0$ **and** $(IBM(i) == 0))$
        $\{i/ = 2; length - -;\}$
    **return** $length;$
$\}$
```

Figure 2: Algorithm to find the length of the longest matching prefix

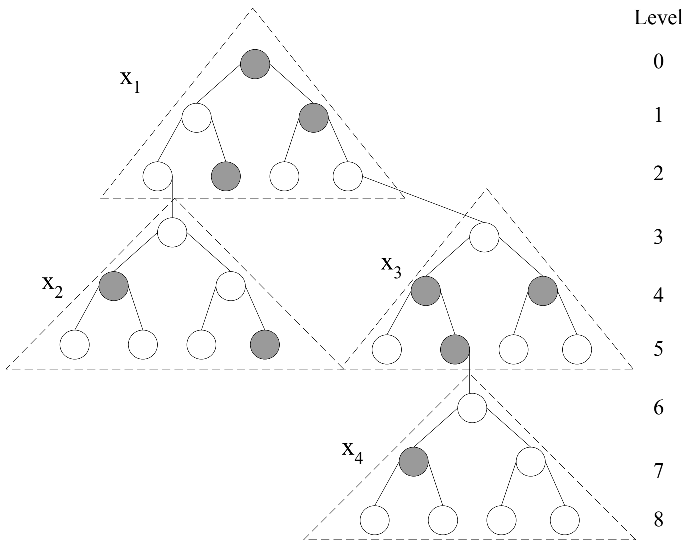

Figure 3: A stride-3 SFE

the root and its left child were also in the original one-bit trie for the prefix set. The remaining 4 nodes were added to complete the partition into a full trie whose height is 2.

Each partition of a stride-$s$ SFE is represented by a $2^s - 1$ bit IBM, a next-hop array whose size is $2^s - 1$, an array of child pointers, and a count of the number of non-null children partitions that this partition has. The array of child pointers can accomodate $2^s$ pointers to children partitions. Figure 4 shows the representation for the stride-3 SFE of Figure 3. Each node of this figure represents a stride-3 partition. The first field of each node is its IBM, the second is the count of children, and the next $2^s$ fields are the children pointers. The next-hop array isn't shown in the figure. Figure 4 defines our dynamic tree bitmap (DTBM) structure. Figure 5 shows the general format for a DTBM node.

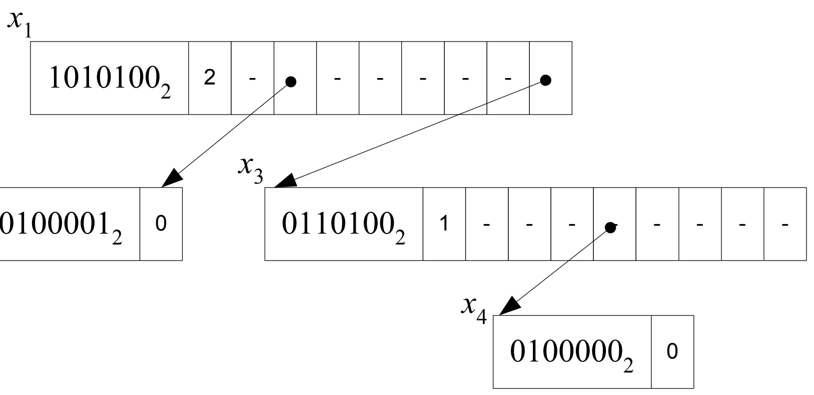

Figure 4: DTBM for Figure 3

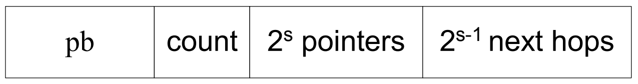

Figure 5: DTBM node format

## 2.2 Lookup

Figure 6 gives the lookup algorithm for a DTBM. *node.pointers*[$i$] returns the $i$th child pointer and *nextHop*[$i$] returns the $i$th entry in the next-hop array of the current DTBM node. We assume that child indexes start at 0 and next hop indexes at 1.

```
**Algorithm** $lookup(d)${// assume there is a matching prefix
   $node = root$;
   **do**{
      $t = node.IBM.llmp(d)$;
      **if** $(t \geq 0)$ // there is a matching prefix in current node
           {$y = node$; $hopIndex = 2^t + d.bits(t)$;}
      $node = node.pointers[d.bits(s)]$;
      shift $d$ left by $s$ bits;
   } **while**($node \neq null$);
   **return** $y.nextHop[hopIndex]$;
}
```

Figure 6: DTBM lookup algorithm

As an example, consider searching the DTBM of Figure 4 using $d = 1110111_2$. The partition represented by node $x_1$ is searched using the first 2 bits (11) of $d$. So, $x_1.IBM.llmp(d)$ returns 1 and $y$ is set to $x_1$ and $hopIndex$ to $2^1 + 1 = 3$. Since $d.bits(s) = d.bits(3) = 7$, $node$ is updated to $x_3$. Also, $d$ is shifted left by $s = 3$ bits and becomes $0111_2$. Next, the partition represented by $x_3$ is searched using the first 2 bits (01) of the updated $d$ and $x_3.IBM.llmp(d)$ returns 2. So, $y$ is updated to $x_3$ and *hopIndex* to 5. This time, $d.bits(3) = 3$ and *node* is updated to $x_4$. Following a left shift by 3 bits, $d$ becomes $1_2$. The search of $x_4$ requires us to first append a 0 to $d$ to obtain $10_2$. Next we search the IBM of $x_4$ beginning at position $2^2 + 2 = 6$. This search returns $-1$ and *node* is updated to *null*. We exit the **do** loop and return the $nextHop[5]$ value of $x_3$.

### 2.3 Inserting a Prefix

Figure 7 gives the algorithm to insert a prefix $p$ into a DTBM whose stride is $s$. In the **while** loop of lines 3 through 6, we walk down the DTBM one node at a time. Each such move consumes $s$ bits of $p$. This walk terminates when either $p$ has fewer than $s$ bits left or the next DTBM node to move to doesn't exist. In the latter case, we continue to sample $p$ $s$-bits at a time adding nodes to the DTBM until we are left with fewer than $s$ bits in $p$. The remaining bits (less than $s$) of $p$ are used to insert $p$ into the last node $Z$ encountered. If $p$ has no remaining bits, the leftmost IBM bit of $Z$ is set to 1 and $Z.nextHop[1]$ is set to the next hop for $p$. Otherwise, bit $2^{p.length} + p.bits(p.length)$ of the IBM of $Z$ is set to 1 and the corresponding *nextHop* entry set to the next hop for $p$.

```
Algorithm insert(p){
01  if (root is null) root = new DTBMNode;
02  node = root;
03  while ($p.length \geq s$ and $node.pointers[p.bits(s)] \neq null$) {
04     node = node.pointers[p.bits(s)];
05     shift p left by s bits;
06  }
07  if ($p.length \geq s$){
08        node.count++;
09        do {
10           node = node.pointers[pbits(s)] = new DTBMNode;
11           node.count = 1;
12           shift p left by s bits;
13        } until ($p.length < s$)
14  }
15  Insert p into node;
}
```

Figure 7: Prefix insertion

## 2.4 Deleting a Prefix

Figure 8 gives the algorithm to delete a prefix $p$ from a DTBM. The algorithm first searches for the DTBM node that contains $p$. During this search, the last sequence of nodes with count field equal to 1 and IBM equal to 0 together with the index of the child pointer used to move from each node in this sequence are stored in the array *nodes*. Whenever the search reaches a DTBM node with count larger than 0 or with a non-zero IBM field, this sequence is reset to *null*. In case $p$ is found in the DTBM, it is removed from its DTBM node $Z$. If now the IBM of $Z$ is 0 and $Z$ has no children, then $Z$ along with its contiguous ancestors with count equal to 1 and IBM equal to 0 are deleted (using *deleteCleanup*).

```
**Algorithm** $delete(p)\{$
  $node = root; i = -1; x = null;$
  **while** $(p.length \geq s$ **and** $node \neq null)$ {
     **if** $(node.count == 1$ **and** $node.IBM == 0)$
          $\{nodes[++i] = node; nodes[++i] = p.bits(s);\}$
     **else** $\{i = -1; x = node; xBits = p.bits(s);\}$
     $node = node.pointers[p.bits(s)];$
     shift $p$ left by $s$ bits;
  }
  **if**$(node \neq null$ **and** $p$ is in $node)\{$
       delete $p$ from $node$;
       **if**$(node.count == 0$ **and** $node.IBM == 0)$
            $deleteCleanup(node, nodes, i, x, xBits)$
  }
}
```

Figure 8: DTBM prefix deletion

## 2.5 Modes of Operation and Analysis

We propose three possible modes of operation for a DTBM–free, packed, and lazy. To analyze the performance of the DTBM data structure in each of these three modes, we assume the data structure will be implemented on a computer with a Cypress FullFlex Dual-Port SRAM. The memory capacity of this SRAM is 36 Mb (512K 72-bit words). The SRAM has two independent ports, both of which support read and write. A 72-bit word may be read/written via either port in 2 cycles (i.e., 8ns using a 250MHz clock). Multiple read/writes may, however, be pipelined and each additional word may be read/written in a single cycle (4ns). We assume that the CPU is sufficiently fast that the time required for a lookup, insert and delete is dominated by the time to read/write data from/to the SRAM. Hence, we analyze

```
Algorithm $deleteCleanup(node, nodes, i, x, xBits)${
    delete $node$;
    while($i \geq 0$){
        $nodes[i-1].pointers[nodes[i]] = null$;
        $nodes[i-1].count = 0$;
        delete $nodes[i-1]$;
        $i- = 2$;
    }
    if($x \neq null$){
        $x.pointers[xBits] = null$;
        $x.count - -$;
    }
    else $root = null$;
}
```

Figure 9: Cleanup algorithm used by $delete(p)$

the complexity of lookup, insert, and delete by counting the (worst-case) number of memory accesses. Note that in our SRAM, two words may be read/written in a single memory access. The reader should note that while the analysis and specific DTBM implementations of this section are very specific to the chosen SRAM, the concepts may be applied easily to any other SRAM.

## 2.6 Free Mode

In this mode, memory management is done using a single chain of free nodes, each node on this chain is of the same size. A node is allocated by removing the first node on this chain and a node is deallocated by adding the deallocated node to the front of this chain. All of memory is initialized to 0 and the way our algorithms work, whenever a node is deallocated, all its children fields also are 0. Consequently, the algorithms work without explicitly setting child pointers to 0 each time a node is allocated. Because of the simple memory management scheme in use, node allocation and deallocation each require a single memory access, that is, 2 cycles.

Since a word of our SRAM is 72 bits in length, the largest stride we can use and yet have an IBM that fits in a word is $s = 6$. The maximum length of an IPv4 prefix is 32. So, with a stride of 6, our DTBM has at most 6 levels. Several choices are possible for the strides of these 6 levels. Each provides a different balance between memory usage and time per operation. We describe one of the many possibilities. The described implementation, called 366666, uses a stride of 3 for the root of the DTBM and a stride of 6 for the remaining DTBM nodes. The levels are numbered 0 (root) through 5. The root of the DTBM requires a 7-bit IBM, 8 child pointers, and 7 next hop fields. The stride-3 DTBM root is represented by storing its IBM in a register, and its next-hop fields in predetermined memory locations. The 8 child pointers are eliminated and instead, we designate nodes numbered 1 through 8 as the (up to) 8 children of the root of the DTBM. We do not store a count field for the root as we shall never delete this node.

A node requires a 63-bit IBM, 6-bit count field, 64 child pointers and 63 next-hop fields. If we allocate 18 bits to each pointer, 256K nodes can be indexed. For the 64 child pointers, we need 16 72-bit words. For each next-hop field, we allocate 12 bits. This is sufficient for 4096 different next-hop possibilities. Hence, to accommodate all the required next-hop fields, we need 11 words. So, each node uses 28 words. Nodes are indexed 1, 2, and so on. By multiplying the node index by 28 and possibly adding an offset, we can determine the first word of the node. With our 18-bit pointer fields, we are, therefore, able to index up to 7M words, which is far greater than the size of the SRAM under consideration.

For a lookup, the move from level 0 (root) to level 1 requires no memory access. At level 1, we access the IBM as well as a child pointer of a single node. The required child pointer is determined by examining $s = 6$ bits of the destination address. From these $s$ bits, knowledge of where the node starts in memory, and knowledge of the node structure, we may compute the index of the word that contains the desired child pointer. Hence, we may obtain the IBM using one port of the SRAM and the child pointer using the other port. So, a total of 2 cycles (or a single memory access) are needed to move from level 1 to level 2. In fact each subsequent move down the TDBM takes the same anount of time. In the worst-case, a lookup requires 4 such moves followed by a read of a level-5 IBM for a total time of 10 cycles. In the end we need another memory access to obtain the next-hop for the longest matching prefix. Hence, the worst-case lookup time is 12 cycles (or 48ns). So, under our assumptions, we can perform almost 21 million lookups per second.

The worst-case for an insert is when the insert requires us to add a new node at each of the levels 2 through 5. For this, we must allocate 4 nodes at a cost of 4 memory accesses or 8 cycles. Additionally, we must update the *count* field and set a child pointer in a node at each of the levels 1 through 4 and set the IBM field and a next-hop field of a level-5 node. Using our dual port memory, we can set the required two fields in each node in 2 cycles. So, an insert requires 18 cycles in the worst-case.

The worst-case for a delete occurs when we delete a prefix that is in a level-5 node and the cleanup step for this deletion, deallocate a node at each of the levels 2 through 5 (note that we reserve nodes 1 through 8 for level 1 and these nodes cannot be deallocated). We reach the desired level-5 node by making 4 reads. Now, we read the IBM of the reached level-5 node, verify that it becomes 0 following the prefix deletion. This requires another read. Four child pointers must be set to 0 and 4 nodes must be deallocated. The writing of a 0 pointer can be done at the same time a node is deallocated. So, a delete requires at most 9 memory access or 18 cycles.

## 2.7 Packed Mode

Although lookups, inserts, and deletes are rather fast in the free mode, the free-mode is wasteful of memory. This is because most of the nodes in a TDBM are leaves and a leaf may be detected by examining its count field (which is 0). Hence a leaf does not require children fields. These fields account for 16 of the 28 words that comprise a node. In an effort to improve the memory efficiency of the TDBM structure, we propose the *packed* mode in which several leaves may be packed into a single 28-word node. We refer to a 28-word node as a type A node.

For the packed mode, we consider a 466656 implementation. This implementation is motivated by the observation that, in practice, IPv4 rule tables have very few prefixes whose length is more than 24. Consequently, a 366666 DTBM is expected to have very few nodes at level 5. So, virtually all of the level-4 nodes are leaves. Further, very few leaves are expected at levels 2 and 3. In the 466656 design, we use type-B nodes at level 5 and type-C nodes at level 4.

In a 466656 TDBM, the root stride is 4, the stride at level 4 is 5, and the stride at every other level is 6. Now, the root of the DTBM requires a 15-bit IBM, 16 child pointers, and 15 next-hop fields. As was the case for our 366666 design, the DTBM root is represented by storing its IBM in a register and its next-hop fields in predetermined memory locations. The 16 child pointers are eliminated and instead, we designate type-A nodes numbered 1 through 16 as the (up to) 16 children of the root of the DTBM. The size of a type-C node is 8 words. A level-4 leaf is represented using a single type-C node. The first word of this node is used to store a 31-bit IBM, a 5-bit count field, an 18-bit pointer (which is null), and a 12-bit next-hop field. The next 6 words are used to store the remaining 30 next-hop fields. The last word is unused. A level-4 non-leaf uses 2 type-C nodes. The first is used as for a level-4 leaf except that the pointer field now points to the second type-C node. The second type-C node stores 32 18-bit child pointers. A type $B$ node requires 12 words and has all the fields of a type A node other than the count field and children pointers. Notice that with 18-bit pointers, we can index only one-fourth as much memory as indexed in free mode. This is because a pointer in a type-A node needs to point to type-C nodes embedded within a type-A node.

Although 3 node-types, each of a different size, are used, memory management is relatively simple. The available memory is partitioned into type A nodes. A type A node may be used as such (i.e., as a type A node) or may be used to house 3 type C nodes (CCC) leaving 4 words unused or to house 2 type C and 1 type B node (CCB). When used in the last two ways, we require that the first 8 words be used as a type C node that has an IBM, count, and next hop. This use leaves the last word in the first type-C node packed into a type-A node (i.e., word 8 of the type-A node) free for control information. Word 8 of a 28-word node is called its *control* word. A child pointer always points to the first word in a type A, B or C node. If words are indexed beginning at 0, then a child pointer is always a multiple of 4 and so its last two bits, which always are 0, need not be stored. Using a 19-bit child pointer, we may, therefore, address up to 1M words. Since a word (or bits in a word) has multiple uses (IBM, child pointer, next hop) it is essential that delete operations set all bits associated with the prefix being deleted to 0. Recall that in the free mode, we didn't set next-hop fields to 0 on deletion nor did we set all IBM bits to 0.

We maintain 7 free lists $free[0:6]$ with the bits in the array index identifying the state of the nodes on each list. So, for example, $free[0] = free[000_2]$ is a list of free nodes in which all 28 words are available for use and $free[3] = free[011_2]$ is a list of free nodes in which the first 8 words are free, the next 8 are used as a type C node, and either the next 8 are used as a type C node or the next 12 as a type B node. List $free[0]$ is a chain and the remaining lists are doubly linked. The *previous* pointer of the first node in each doubly linked list may be set to any arbitrary value. 3 bits of the control word of a node into which we have packed type B or C nodes are used to indicate which list the node is in (these 3 bits are 0 in case the node is in no list) and 18 bits for each of the pointers *previous* and *next* needed for a doubly linked list.

Type A nodes (those needed at levels 2 and 3) may be allocated only from the list $free[0]$. Type $C$ nodes that are to have an IBM are preferentially allocated from any one of the lists $free[1:6]$ and remaining type C nodes are preferentially allocated from the lists $free[1:2]$ and $free[4:6]$. Type $B$ nodes are preferentially allocated from any of the free lists $free[010_2]$, $free[100_2]$, and $free[110_2]$. In case a type B or C node cannot be allocated from one of its preferential lists, the allocation is made from $free[0]$. Note that when a type B node is allocated from the list $free[0]$, the allocated node is removed from the front of the list $free[0]$ and put on to the list $free[001_2]$ and so on. Suppose node $X$, which is at the front of $free[100_2]$ is to be allocated for use as a type B node. We need 1 memory access to remove this node from $free[100_2]$ as we must reset $free[100_2]$ to the next node on the list (this requires us to read the control word of $X$). Note that the control word of the new node at the front of $free[100_2]$ is not read or written as its *previous* pointer need not be set to *null*. Next, $X$ is to be added to the front of $free[101_2]$. For this, we set the *next* field of the control word of $X$ to $free[101_2]$; write this control word and read the control word of $free[101_2]$ (the read and write are done in the same memory access using both memory ports). The *previous* pointer in the control word of $free[101_2]$ is changed to $X$ and written back to memory. So, 2 memory accesses are needed to add $X$ to the front of $free[101_2]$. The worst-case memory accesses for a node allocation is 3, a cost of 6 cycles. More precisely, 1 memory access is needed to allocate a node of type A and up to 3 to allocate a node of type B or C.

To deallocate a node from levels 2 and 3, we simply add the deallocated node to the front of $free[0]$. This has a cost of 1 memory access (recall that $free[0]$ is a chain). To deallocate any other node we must first compute the start of the 28-word node in which it is housed. Let $x$ be the start of a type $B$ or type $C$ node. This node is housed in a type A node that begins at $y = \lfloor x/28 \rfloor * 28$. We read the control word of the type A node that begins at $y$ and compute the new status for this node. In the worst case, this type A node will have to be moved from the middle of one doubly linked free list to the front of another. From the control word of $y$ we determine its current previous and next nodes. The control words of these previous and next nodes may be read and modified to reflect the deletion of $y$ with 2 memory accesses using our dual-port memory. Another up to 2 memory accesses are needed to add $y$ to the front of its new free list. So, deallocating a node of type B or C may make up to 5 accesses; a node of type A may be deallocated with a single memory access.

The worst-case lookup cost in the 466656 scheme is 2 cycles more than in the 366666 scheme, because at level 4 we cannot access the IBM and needed child pointer concurrently. We must first access a type-C node using a child pointer in a level-3 node, then use the pointer in this type-C node to determine the word in the next type-C node that contains the child pointer to the desired level-5 node. So, the total lookup time is 14 cycles.

The worst-case for an insert is when the insert requires us to add a new node at each of the levels 2 through 5. For this, we must allocate 2 type A nodes and 3 type B or type C nodes. Since an allocation from $free[0]$ can be done during the first read access of an allocation from any of the other lists, the worst-case accesses needed to allocate the needed 5 nodes is 9. Additionally, we must update the *count* field and set a child pointer in a node at each of the levels 1 through 3, set the *count* and *next* field of a type C node at level 4, set a child pointer in a level-4 type-C node, and set the IBM field and a next-hop field of a level-5 node. Using our dual port memory, we can set the required fields in each of these 6 nodes in 2 cycles. So, an insert requires 30 (15 memory accesses) cycles in the worst-case.

The worst-case for a delete occurs when we delete a prefix that is in a level-5 node and the cleanup step for this deletion, deallocate a type A node at each of the levels 2 and 3, 2 type C nodes at level 4 and a type B node at level 5 . These node deallocations have a nominal worst-case cost of 17 memory accesses (34 cycles). The worst-case cost can be reduced to 15 memory accesses by deallocating the type A nodes in slots in which the deallocate of the remaining 3 nodes uses only 1 port of our dual port memory. We reach the desired level-5 node by making 5 reads (note that at level 4, we need an extra read to access the appropriate child pointer). Now, we read the IBM of the reached level-5 node, verify that it becomes 0 following the prefix deletion, write out the 0 IBM, set the next-hop field for the deleted prefix to 0, and set the *next* pointer in a level-4 type-C node to 0. This requires another 2 accesses. A child pointer needs to be set to *null* in 3 of the 5 nodes being deallocated as well as in a level-1 node. These 4 child pointers can be set to *null* with 2 memory accesses. The total number of memory access is 24 for a cycle count of 48.

As we can see, our attempt to conserve memory has resulted in a significant increase in the worst-case cost of an insert and delete.

### 2.7.1 Lazy Mode

In this mode, prefix deletion is done by simply setting the appropriate bit of the IBM of the node that contains the deleted prefix to 0. The cleanup action performed by *deleteCleanup* is not done. Consequently, nodes are never deallocated. While it is easy to see that lazy mode operation, through its elimination of the cleanup action, supports faster deletes, lazy mode operation also speeds inserts as memory management is simplified. Besides a reduction in the worst-case time for an insert, average-case time also may be reduced as future inserts may reuse nodes that would, otherwise, have been deallocated during the cleanup phase of a delete. The lazy mode of operation, however, runs the risk of failing to make an insert that would have succeeded had we freed nodes as is done by *deleteCleanup*. To avoid this outcome, we propose rebuilding the data structure whenever we get “close” to running out of memory. The rebuild time, even for large router-tables, is of the order of tens of milliseconds.

In the lazy mode of operation, we do not require a *count* field (Figure 5), as this field is used only to assist in determining which nodes are to be deallocated in the cleanup phase of *delete*. Figures 10 and 11 give the algorithms for insert and delete in the lazy mode of operation.

We evaluate lazy-mode implementations of both 366666 and 466656 DTBMs. For 366666, we use nodes of type A for levels 1 through 4 and of type B for level 5. The root of the DTBM is stored as for the non-lazy implementation; type A nodes numbered 1 through 8 are used for the (up to) 8 children of the root of the DTBM.

Although we no longer use a *count* field, the size of type A and type B nodes is unchanged. That

```
Algorithm $lazyInsert(p)${
    if ($root$ is $null$) $root$ = new DTBMNode;
    $node = root$;
    while ($p.length \geq s$ and $node.pointers[p.bits(s)] \neq null$) {
        $node = node.pointers[p.bits(s)]$;
        shift $p$ left by $s$ bits;
    }
    if ($p.length \geq s$){
        do {
            $node = node.pointers[pbits(s)]$ = new DTBMNode;
            shift $p$ left by $s$ bits;
        } until ($p.length < s$)
    }
    Insert $p$ into $node$;
}
```

Figure 10: Prefix insertion for lazy mode

```
Algorithm $lazyDelete(p)${
    $node = root$;
    while ($p.length \geq s$ and $node \neq null$) {
        $node = node.pointers[p.bits(s)]$;
        shift $p$ left by $s$ bits;
    }
    if($node \neq null$ and $p$ is in $node$)
        delete $p$ from $node$;
}
```

Figure 11: Prefix deletion for lazy mode

is, a type A node is 28 words long and a type B node is 12 words. As nodes are never deallocated in the lazy mode of operation and as a type A node never becomes a type B node (or the reverse) as the result of an insert or delete, a very simple memory management scheme may be used. The memory to be managed is a block of contiguous words. All bits in this block are initialized to zero. Type A nodes are allocated from one end of this block and type B nodes from the other end. When a node is allocated, it is assigned an index. Type A nodes are indexed 1, 2, and so on. Type B nodes are indexed in the same way. A child pointer in a type A node stores the index of the node to which it points. When navigating through the DTBM, the node type is inferred from the level we currently are at and the first word in the node is obtained by multiplying the node index by its size (28 or 12) and adding or subtracting the result to/from a base address. Since each node is at least 12 words long, fewer than 44K nodes are possible in our SRAM.

The lookup time is the same, 12 cycles, as that for a non-lazy 366666 DTM. The worst-case for an insert is when the new prefix goes into either a new or existing level-5 node. Consider the former case. Such an insert requires us to read a pointer field from one node at each of levels 1 through 4 (alternatively, at each level where there is no node to read, a new node is created and a pointer written to the parent node), write a pointer field of a level 4 node, and write a new level-5 node. The write of the level-5 node is done by writing the word that contains its IBM plus the word that has the next-hop value for the inserted prefix. The remaining words are unchanged (recall that all bits were set to 0 initially). Both words of the level-5 node that are to be written may be written in parallel using our 2-ported memory. So, a worst-case insert requires 4 reads from and 2 writes to the 2-ported memory for a total time of 12 cycles.

The worst-case for a delete occurs when we delete a prefix that is in a level-5 node. We reach the desired level-5 node by making 4 reads. Now, we must set a bit of the IBM of the reached level-5 node to 0 (there is no need to set the corresponding next-hop value to *null*). To make this change, we must read the old IBM, change it, and write the changed IBM back to memory. This requires an additional read and write access. So, we need 5 read and 1 write accesses for a total of 12 cycles. Interestingly, all 3 of our operations have the same worst-case requirement of 12 cycles!

The root of a lazy-mode 466656 DTBM is represented as for its non-lazy counterpart and we designate type A nodes numbered 1 through 16 as the (up to) 16 children of the root of the DTBM. Type A nodes are used at levels 1, 2, and 3 and type C nodes at levels 4 and 5. The size of a type C node is 8 words. At level 5, we use two adjacent type-C nodes for each DTBM node. Since the nodes are adjacent, no pointer is needed from the first node to the second. This use of 2 type-C nodes wastes 4 words per DTBM level-5 node. However, since very few nodes are expected at this level, the total waste is expected to be small.

Since the lazy-mode 466656 design uses only 2 node sizes, we may use the same memory management scheme as used in the lazy-mode 366666 scheme. The worst-case lookup cost in the 466656 scheme is the same, 14 cycles, as for the non-lazy mode.

The worst-case for an insert is when we insert a prefix into an existing level-5 node. It takes 10 cycles to get to this level-5 node, 2 cycles to read this level-5 node's IBM, and another 2 cycles to write the modified IBM as well as the next-hop associated with the inserted prefix. The total insert time is 14 cycles.

A worst-case delete is one that deletes a prefix from a level-5 node. As for a worst-case insert, it takes

10 cycles to get to this level-5 node, 2 to read its IBM and another 2 to write the changed IBM. The total time again is 14 cycles.

Although the lazy-mode 466656 design takes 2 cycles more per operation than does the lazy-mode 366666 design, we expect the 466656 design to require less memory to represent real-world IPv4 router tables.

## 3 Experimental Results

We compare the DTBM structure with two other structures–BaRT [18, 19] and TBM [1]–that have been proposed for dynamic router tables. For BaRT, we use the versions 8888, 86558 and 844448 reported on in [18] for stand-alone mode and for TBM, we use the 13-4-4-4-4-3 software reference design of [1]. For our analysis, we assume that each memory accesses requires two cycles. Although the Cypress FullFlex Dual-Port SRAM is able to pipeline memory accesses so that all but the first access take one cycle each, none of the data structures we consider is able to use this feature and the number of cycles becomes twice the number of memory accesses. Table 1 gives the worst-case number of cycles needed for a lookup and update for the considered data structures. As can be seen, the DTBM variants have considerable better update characteristics; all are comparable on lookup cycles.

| | Lookup | Update | Memory (Mbits) | | | | | |
|---|---|---|---|---|---|---|---|---|
| | (cycles) | (cycles) | Paix1 | Pb1 | MaeWest | Aads | Pb2 | Paix2 |
| Table Size | | | 16,172 | 22,225 | 28,889 | 31,827 | 35,302 | 85,987 |
| BaRTs 8-8-8-8 | 12 | 794[(a)] | 4.5 | 5.3 | 7.3 | 8.5 | 9.0 | 21.5 |
| BaRTs 8-6-5-5-8 | 14 | 226[(a)] | 1.5 | 1.9 | 2.5 | 2.8 | 2.9 | 6.8 |
| BaRTs 8-4-4-4-4-8 | 16 | 90[(a)] | 1.2 | 1.6 | 2.0 | 2.3 | 2.5 | 5.3 |
| TBM 13-4-4-4-4-3 | 12 | $> 6000$ | 1.2 | 1.5 | 1.8 | 1.9 | 2.3 | 4.2 |
| FreeMode 3-6-6-6-6-6 | 12 | 18 | 18 | 26 | 32 | 35 | 26 | 36 |
| LazyMode 3-6-6-6-6-6 | 12 | 12[(b)] | 18 | 26 | 32 | 34 | 26 | 36 |
| PackedMode 4-6-6-6-5-6 | 14 | 48 | 12 | 17 | 20 | 22 | 17 | 23 |
| LazyMode 4-6-6-6-5-6 | 14 | 14[(b)] | 12 | 16 | 19 | 21 | 16 | 21 |

(a). Excludes memory accesses for storage management.
(b). Data structure needs to be rebuilt when we get close to running out of memory.

Table 1: Memory requirements and the worst-case lookup and update times

Table 1 gives also the memory required by the different data structures for 6 publically available router tables. These databases were obtained from [11]. The databases Paix1, Pb1, MaeWest and Aads were obtained on Nov. 22, 2001, while Pb2 and Paix2 were obtained on Sep. 13, 2000. These 6 databases ranged in size from a low of 16,172 (Paix1) rules to a high of 85,987 (Paix2) rules. Although our TDBM scheme takes more memory than the BaRT and TBM schemes, the memory required is acceptable and within the bounds of current technology.

## 4 Conclusion

We have proposed a variant, DTBM, of the tree bitmap, TBM, data structure of [1]. This variant greatly simplifies memory management and supports a much higher rate of update than supported by the original TBM structure. Although this improvement comes at the expense of increased memory requirement, the memory required by DTBM is within the constraints of current technology. The DTBM outperforms also the BaRT [18, 19] structure but, again, at the expense of increased memory requirement.

## References

[1] W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software IP lookups with incremental updates, *ACM SIGCOMM Computer Communication Review*, 34, 2, April 2004.

[2] F. Baboescu, S. Singh, and G. Varghese, Packet classification for core routers: is there an alternative to CAMs? *IEEE INFOCOM*, 2003.

[3] A. Basu and G. Narlika, Fast incremental updates for pipelined forwarding engines, *IEEE INFOCOM*, 2003.

[4] T. Cormen, C. Lieserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Second Edition, 2001.

[5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small forwarding tables for fast routing lookups, *ACM SIGCOMM*, 1997, 3-14.

[6] W. Doeringer, G. Karjoth, and M. Nassehi, Routing on longest-matching prefixes, *IEEE/ACM Transactions on Networking*, 4, 1, 1996, 86-97.

[7] F. Ergun, S. Mittra, S. Sahinalp, J. Sharp, and R. Sinha, A dynamic lookup scheme for bursty access patterns, *IEEE INFOCOM*, 2001.

[8] P. Gupta, S. Lin, and N. McKeown, Routing lookups in hardware at memory access speeds, *IEEE INFOCOM*, 1998.

[9] P. Gupta and N. McKeown, Dynamic algorithms with worst-case performance for packet classification, *IFIP Networking*, 2000.

[10] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of Data Structures in C++, W.H. Freeman, NY, 1995, 653 pages.

[11] Merit, Ipma statistics, http://nic.merit.edu/ipma.

[12] C. Labovitz, G. Malan, and F. Jahanian, Internet routing instability, *IEEE/ACM Transactions on Networking*, 1997.

[13] B. Lampson, V. Srinivasan, and G. Varghese, IP lookup using multi-way and multicolumn search, *IEEE INFOCOM 98*, 1998.

[14] H. Lu and S. Sahni, $O(\log n)$ dynamic router-tables for prefixes and ranges. *IEEE Symposium on Computers and Communications*, 2003.

[15] H. Lu and S. Sahni, Enhanced interval trees for dynamic IP router-tables. *IEEE Transactions on Computers*, 53, 12, 2004, 1615-1628.

[16] H. Lu, K. Kim, and S. Sahni, Prefix- and interval-partitioned dynamic IP router-tables. *IEEE Transactions on Computers*, 54, 5, 2005, 545-557.

[17] H. Lu and S. Sahni, A B-tree router-table design. *IEEE Transactions on Computers*, 54, 7, 2005, 813-824.

[18] J. Lunteren, Searching very large routing tables in fast SRAM, *Proceedings ICCCN*, 2001.

[19] J. Lunteren, Searching very large routing tables in wide embedded memory, *Proceedings Globecom*, 2001.

[20] C. Macian, and R. Finthammer, An evaluation of the key design criteria to achieve high update rates in packet classifiers, *IEEE Network*, Nov/Dec.24-29, 2001.

[21] C. Matsumoto, CAM vendors consider algorithmic alternatives *EETIMES*, May 2002.

[22] A. J. McAuley, and P. Francis, Fast routing table lookup using CAMs, *IEEE INFOCOM*, 1993.

[23] S. Nilsson and G. Karlsson, Fast address look-up for Internet routers, *IEEE Broadband Communications*, 1998.

[24] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network*, 2001, 8-23.

[25] S. Sahni and K. Kim, Efficient construction of fixed-Stride multibit tries for IP lookup, *Proceedings 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, 2001.

[26] S. Sahni and K. Kim, Efficient construction of variable-stride multibit tries for IP lookup, *Proceedings IEEE Symposium on Applications and the Internet (SAINT)*, 2002, 220-227.

[27] S. Sahni and K. Kim, $O(\log n)$ dynamic packet routing, *IEEE Symposium on Computers and Communications*, 2002.

[28] S. Sahni and K. Kim, Efficient dynamic lookup for bursty access patterns, submitted.

[29] S. Sahni, Data structures, algorithms, and applications in Java, McGraw Hill, NY, 2000, 833 pages.

[30] S. Sahni, K. Kim, H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, 2002, 3-14.

[31] D. Shah, and P. Gupta, Fast updating algorithms for TCAMs, *IEEE MICRO*, 21, 1, 2001, 36-47.

[32] K. Sklower, A tree-based routing table for Berkeley Unix, Technical Report, University of California, Berkeley, 1993.

[33] V. Srinivasan and G. Varghese, Faster IP lookups using controlled prefix expansion, *ACM Transactions on Computer Systems*, Feb:1-40, 1999.

[34] S. Suri, G. Varghese, and P. Warkhede, Multiway range trees: Scalable IP lookup with fast updates, *GLOBECOM 2001.*

[35] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, Scalable high speed IP routing lookups, *ACM SIGCOMM*, 1997, 25-36.