Succinct Representation Of Static Packet Forwarding Tables *

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

November 14, 2006

Abstract

We develop algorithms for the compact representation of the trie structures that are used for Internet packet forwarding. Our compact representations are experimentally compared with competing compact representations for packet forwarding tables and found to simultaneously reduce the number of memory accesses required for a lookup as well as the memory required to store the forwarding table.

1 Introduction

An Internet router makes the forwarding decisions for incoming packets based on their header fields using a forwarding table, which is comprised of rules of the form $(P, Nexthop)$. P is a prefix that specifies a range of destination IP addresses, and if an incoming packet's destination address matches P, the associated $Nexthop$ specifies where to forward it next. In commercial forwarding table, an incoming packet may match several rules. In that case, the $Nexthop$ associated to the matching rule with the longest prefix is used for the forwarding. Therefore, a router is required to find the longest matching prefix from a table of prefixes to make the forwarding decision. Data structures for longest-prefix matching have been extensively studied (see [15, 16], for surveys).

Our focus in this paper is the succinct representation of the trie structures that are commonly used to represent forwarding tables; the succinct representation is to support high-speed packet forwarding. For this work, we assume that the forwarding table is static. That is, the set of rules that comprise the forwarding table does not change (no inserts/deletes). This assumption is consistent with that made in most of the forwarding table literature where the objective is to develop a memory-efficient forwarding table representation that can be searched very fast.

We begin, in Section 2, by reviewing the binary trie representation of a forwarding table together with the research that has been done on the succinct representation of these structures. In Sections 3 through 5, we develop our algorithms for the succinct representation of trie structures. Experimental results are presented in Section 6.

2 Background and Related Work

Many of the data structures developed for the representation of a forwarding table are based on the *binary trie* structure [4]. A binary trie is a binary tree structure in which each node has a data field and two children fields.

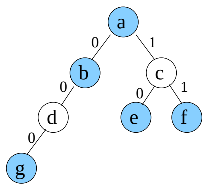
*This research was supported, in part, by the National Science Foundation under grant ITR-0326155

Branching is done based on the bits in the search key. A left child branch is followed at a node at level i (the root is at level 0) if the ith bit of the search key (the leftmost bit of the search key is bit 0) is 0; otherwise a right child branch is followed. Level i nodes store prefixes whose length is i in their data fields. The node in which a prefix is to be stored is determined by doing a search using that prefix as key. Let N be a node in a binary trie. Let $Q(N)$ be the bit string defined by the path from the root to N. $Q(N)$ is the prefix that corresponds to N. $Q(N)$ is stored in $N.data$ in case $Q(N)$ is one of the prefixes to be stored in the trie.

Fig. 1 (a) shows a set of 5 prefixes. The $*$ shown at the right end of each prefix is used neither for the branching described above nor in the length computation. So, the length of $P2$ is 1. Fig. 1 (b) shows the binary trie corresponding to this set of prefixes. Shaded nodes correspond to prefixes in the rule table and each contains the next hop for the associated prefix. The binary trie of Fig. 1 (b) differs from the 1-bit trie used in [17], [16], and others in that a 1-bit trie stores up to 2 prefixes in a node (a prefix of length l is stored in a node at level $l-1$) whereas each node of a binary trie stores at most 1 prefix. Because of this difference in prefix storage strategy, a binary trie may have up to 33 (129) levels when storing IPv4 (IPv6) prefixes while the number of levels in a 1-bit trie is at most 32 (128).

P1	*
P2	0*
P3	000*
P4	10*
P5	11*

(a) 5 prefixes



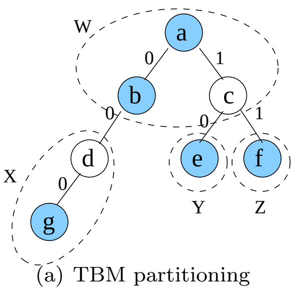
(b) Corresponding binary trie

Figure 1: Prefixes and corresponding binary trie

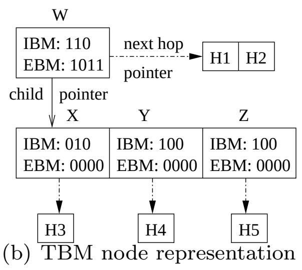
For any destination address d, we may find the longest matching prefix by following a path beginning at the trie root and dictated by d. The last prefix encountered on this path is the longest prefix that matches d. While this search algorithm is simple, it results in as many cache misses as the number of levels in the trie. Even for IPv4, this number, which is at most 33, is too large for us to classify/forward packets at line speed. Several strategies–e.g., LC trie [12], Lulea [2], tree bitmap [3], multibit tries [17], shape shifting tries [18]–have been proposed to improve the lookup performance of binary tries. All of these strategies collapse several levels of each subtree of a binary trie into a single node, which we call a *supernode*, that can be searched with a number of memory accesses that is less than the number of levels collapsed into the supernode. For example, we can access the correct child pointer (as well as its associated prefix) in a multibit trie with a single memory access independent of the size of the multibit node. The resulting trie, which is composed of supernodes, is called a *supernode trie.* Lunteren [9, 10] has devised a perfect-hash-function scheme for the compact representation of the supernodes of a multibit trie.

The data structure we propose in this paper also is a supernode trie structure. Our structure is most closely related to the shape shifting trie (SST) structure of Song et al. [18], which in turn draws heavily from the tree bitmap (TBM) scheme of Eatherton et al. [3] and the technique developed by Jacobson [6, 11] for the succinct representation of a binary tree. In TBM we start with the binary trie for our forwarding table and partition this binary trie into subtries that have at most S levels each. Each partition is then represented as a (TBM) supernode. S is the *stride* of a TBM supernode. While $S = 8$ is suggested in [3] for real-world IPv4 forwarding tables, we use $S = 2$ here to illustrate the TBM structure.

Fig. 2 (a) shows a partitoning of the binary trie of Fig. 1 (b) into 4 subtries W–Z that have 2 levels each. Although a full binary trie with $S = 2$ levels has 3 nodes, X has only 2 nodes and Y and Z have only one node each. Each partition is is represented by a supernode (Fig. 2 (b)) that has the following components:



(a) TBM partitioning



(b) TBM node representation

Figure 2: TBM for binary trie of Fig. 1 (b)

1. A $(2^S - 1)$-bit bit map IBM (internal bitmap) that indicates whether each of the up to $2^S - 1$ nodes in the partition contains a prefix. The IBM is constructed by superimposing the partition nodes on a full binary trie that has S levels and traversing the nodes of this full binary trie in level order. For node W, the IBM is 110 indicating that the root and its left child have a prefix and the root's right child is either absent or has no prefix. The IBM for X is 010, which indicates that the left child of the root of X has a prefix and that the right child of the root is either absent or has no prefix (note that the root itself is always present and so a 0 in the leading position of an IBM indicates that the root has no prefix). The IBM's for Y and Z are both 100.

2. A 2^S-bit EBM (external bit map) that corresponds to the 2^S child pointers that the leaves of a full S-level binary trie has. As was the case for the IBM, we superimpose the nodes of the partition on a full binary trie that has S levels. Then we see which of the partition nodes has child pointers emanating from the leaves of the full binary trie. The EBM for W is 1011, which indicates that only the right child of the leftmost leaf of the full binary trie is null. The EBMs for X, Y and Z are 0000 indicating that the nodes of X, Y and Z have no children that are not included in X, Y, and Z, respectively. Each child pointer from a node in one partition to a node in another partition becomes a pointer from a supernode to another supernode. To reduce the space required for these inter-supernode pointers, the children supernodes of a supernode are stored sequentially from left to right so that using the location of the first child and the size of a supernode, we can compute the location of any child supernode.

3. A child pointer that points to the location where the first child supernode is stored.

4. A pointer to a list NH of next-hop data for the prefixes in the partition. NH may have up to $2^S - 1$ entries. This list is created by traversing the partition nodes in level order. The NH list for W is $nh(P1)$ and $nh(P2)$, where $nh(P1)$ is the next hop for prefix $P1$. The NH list for X is $nh(P3)$. While the NH pointer is part of the supernode, the NH list is not. The NH list is conveniently represented as an array.

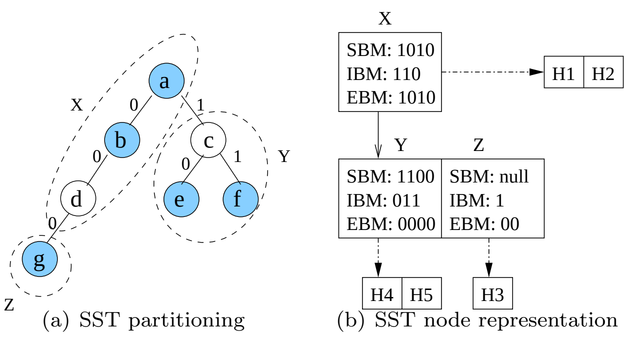
The NH list (array) of a supernode is stored separate from the supernode itself and is accessed only when the longest matching prefix has been determined and we now wish to determine the next hop associated with this prefix. If we need b bits for a pointer, then a total of $2^{S+1} + 2b - 1$ bits (plus space for an NH list) are needed for each TBM supernode. Using the IBM, we can determine the longest matching prefix in a supernode; the EBM is used to determine whether we should move next to the first, second, etc. child of the current supernode. If a single memory access is sufficient to retrieve an entire supernode, we can move from one supernode to its child with a single access. The total number of memory accesses to search a supernode trie becomes the number of levels in the supernode trie plus 1 (to access the next hop for the longest matching prefix).

The SST supernode structure proposed by Song et al. [18] is obtained by partitioning a binary trie into subtries that have at most K nodes each. K is the *stride* of an SST supernode. To correctly search an SST, each SST supernode requires a shape bit map (SBM) in addition to an IBM and EBM. The SBM used by Song et al. [18] is the succinct representation of a binary tree developed by Jacobson [6]. Jacobson's SBM is obtained by replacing every null link in the binary tree being coded by the SBM with an external node. Next, place a 0 in every external node and a 1 in every other node. Finally, traverse this extended binary tree in level order, listing the bits in the nodes as they are visited by the traversal.

Suppose we partition our example binary trie of Fig. 1 (b) into binary tries that have at most $K = 3$ nodes each. Fig. 3 (a) shows a possible partitioning into the 3 partitions X-Z. X includes nodes a, b and d of Fig. 1 (b); Y includes nodes c, e and f; and Z includes node g. The SST representation has 3 (SST) supernodes. The SBMs for the supernodes for X-Z, respectively, are 1101000, 1110000, and 100. Note that a binary tree with K internal nodes has exactly $K + 1$ external nodes. So, when we partition into binary tries that have at most K internal nodes, the SBM is at most $2K + 1$ bits long. Since the first bit in an SBM is 1 and the last 2 bits are 0, we don't need to store these bits explicitly. Hence, an SBM requires only $2K - 2$ bits of storage. Fig. 3 (b) shows the node representation for each partition of Fig. 3 (a). The shown SBMs exclude the first and last two bits.

The IBM of an SST supernode is obtained by traversing the partition in level order; when a node is visited, we ouput a 1 to the IBM if the node has a prefix and a 0 otherwise. The IBMs for nodes X-Z are, respectively, 110, 011, and 1. Note than the IBM of an SST supernode is at most K bits in length.

To obtain the EBM of a supernode, we start with the extended binary tree for the partition and place a 1 in each external node that corresponds to a node in the original binary trie and a 0 in every other external node. Next, we visit the external nodes in level order and output their bit to the EBM. The EBMs for our 3 supernodes are, respectively, 1010, 0000, and 00. Since the number of external nodes for each partition is at most $K + 1$, the



(a) SST partitioning (b) SST node representation

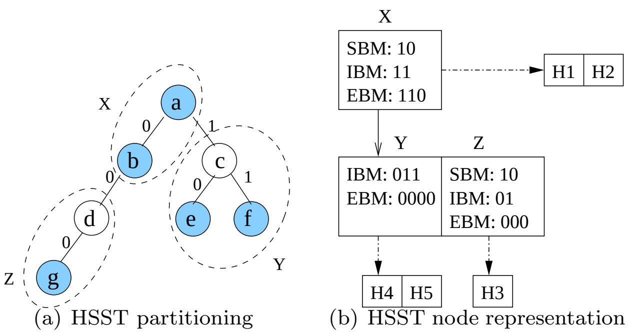
Figure 3: SST for binary trie of Fig. 1 (b)

size of an EBM is at most $K + 1$ bits.

As in the case of the TBM structure, child supernodes of an SST supernode are stored sequentially and a pointer to the first child supernode maintained. The NH list for the supernode is stored in separate memory and a pointer to this list maintained within the supernode. Although the size of an SBM, IBM and EBM varies with the partition size, an SST supernode is of a fixed size and allocates $2K$ bits to the SBM, K bits to the IBM and $K + 1$ bits to the EBM. Unused bits are filled with 0s. Hence, the size of an SST supernode is $4K + 2b - 1$ bits.

Song et al. [18] develop an $O(m)$ time algorithm, called *post-order pruning*, to construct a minimum-node SST, for any given K, from an m-node binary trie. They develop also a *breadth-first pruning* algorithm to construct, for any given K, a minimum height SST. The complexity of this algorithm is $O(m^2)$.

For dense binary tries, TBMs are more space efficient than SSTs. However, for sparse binary tries, SSTs are more space efficient. Song et al. [18] propose a hybrid SST (HSST) in which dense subtries of the overall binary trie are partitioned into TBM supernodes and sparse subtries into SST supernodes. Fig. 4 shows an HSST for the binary trie of Fig. 1 (b). For this HSST, $K = S = 2$. The HSST has two SST nodes X and Z, and one TBM node Y.



(a) HSST partitioning (b) HSST node representation

Figure 4: HSST for binary trie of Fig. 1(b)

Although Song et al. [18] do not develop an algorithm to construct a space-optimal HSST, they propose a heuristic that is a modification of their breadth-first pruning algorithm for SSTs. This heuristic guarantees that the height of the constructed HSST is no more than that of the height-optimal SST.

3 Minimum-Height SSTs

The breadth-first pruning algorithm of Song et al. [18] constructs, for any given K and binary trie T, a minimum height SST. The complexity of this algorithm is $O(m^2)$, where m is the number of nodes in T. In this section, we develop an $O(m)$ algorithm for this task. Our algorithm, which we call $minHtSST$, performs a postorder traversal[1] of T. When a node x of T is visited during this traversal, one or both of the currently remaining subtries of x and, at times, even the entire remaining subtrie rooted at x may be pruned off to form a node of the SST being constructed.

When $minHtSST$ visits a node x of T, some (or all) of the descendents of x in T have been pruned by earlier node visits. The pruned descendents of x have been mapped into supernodes that form one or more SSTs. These SSTs are referred to as the *SSTs that hang from* x. Some of these SSTs that hang from x were created during visits of nodes in the left subtree of x. These SSTs are called the *left hanging* SSTs; the remaining SSTs are the *right hanging* SSTs of x. We use the following notation: $x.leftChild$ ($x.rightChild$) is the left (right) child of x in T; $x.st$ is the set of nodes in the subtrie of T rooted at x; $x.rn$ (remaining nodes) is the subset of $x.st$ that have not been pruned off at the time x is visited; $x.size$ is the number of nodes in $x.rn$; $x.SSTs$ is the set of SSTs that hang from x at the time x is visited; $x.leftSSTs$ ($x.rightSSTs$) is the subset of $x.SSTs$ that are left (right) hanging SSTs. $x.lht = -1$ (left height) if $x.leftSSTs$ is empty. Otherwise, $x.lht$ is the maximum height of an SST in $x.leftSSTs$ (the height of an SST is 1 less than the number of levels in the tree). $x.rht$ is the corresponding quantity for the $x.rightSSTs$ and $x.ht = max\{x.lht, x.rht\}$.

The function $prune(y)$ prunes T at the node y by removing all nodes in $y.rn$. The nodes in $y.rn$ are used to create a supernode whose subtries are $y.SSTs$. When y is NULL, $prune(y)$ is a NULL operation. Fig. 5 gives the visit function employed by our postorder traversal algorithm $minHtSST$. x is the node of T being visited. To avoid code clutter, we do not show the code needed to update $size$, lht, rht, $SSTs$, and so on. This visit function has 3 mutually exclusive cases. Exactly one of these is executed during a visit.

It is easy to see that if T is traversed in postorder using the visit function of Fig. 5, then $x.leftChild.size < K$ and $x.rightChild.size < K$ when x is visited. Less evident is the fact that when x is visited, every node y that is in the left (right) subtree of x and in $x.rn$ has $y.ht = x.lht$ ($x.rht$).

Theorem 1 *For every binary trie T and integer $K > 0$, the postorder traversal algorithm $minHtSST$ constructs an SST that has minimum height.*

Proof See [8]. ■

Since the visit function of Fig. 5 can be implemented to run in $O(1)$ time, the complexity of our postorder traversal function $minHtSST$ is $O(m)$ where m is the number of nodes in the binary trie T. Note that the number

[1]Our postorder traversal algorithm is quite different from the postorder traversal algorithm POP of [18], which constructs an SST that has the fewest number of nodes.

```
Case 1: [$x.lht == x.rht$]
    if ($x.size > K$)
    {$prune(x.leftChild)$; $prune(x.rightChild)$; }
    else if ($x.size == K$) $prune(x)$;
    return;

Case 2: [$x.lht < x.rht$]
    $prune(x.leftChild)$;
    update $x.size$;
    if ($x.size == K$) $prune(x)$;
    return;

Case 3: [$x.lht > x.rht$]
    Symmetric to Case 2.
```

Figure 5: Visit function for $minHtSST$

of nodes in the binary trie for n prefixes whose length is at most W is $O(nW)$. So, in terms of n and W, the complexity of $minHtSST$ is $O(nW)$.

4 Space-Optimal HSSTs

Let $minSpHSST(T,H)$ be a minimum space HSST for the binary trie T under the restrictions that the stride of the TBM nodes is S and that of the SST nodes is K and the height of the HSST is at most H. We assume that S and K are such that the size of a TBM supernode is the same as that of an SST supernode. Although it may not be possible to choose S and K so that the number of bits needed by a TBM supernode is exactly equal to that needed by an SST supernode, in practice, node size is chosen to match the bandwidth of the memory we have. This means that we waste a few bits in every supernode, if necessary, to ensure a supernode size equal to the memory bandwidth. So, in practice, with the wasted memory factored in, the size of a TBM supernode equals that of an SST supernode. Hence, minimizing the space required by an HSST is equivalent to minimizing the number of supernodes in the HSST. Therefore, we use the number of supernodes in an HSST as a measure of its space requirement.

Let $ST(N)$ denote the subtree of T that is rooted at node N. So, $T = ST(root(T))$. Let $opt(N,h)$ be the number of supernodes in $minSpHSST(ST(N),h)$. $opt(root(T),H)$ is the number of supernodes in $minSpHSST(T,H)$. We shall develop a dynamic programming recurrence for $opt(N,h)$. This recurrence may be solved to determine $opt(root(T),H)$. A simple extension to the recurrence enables us to actually compute $minSpHSST(T,H)$.

Let $opt(N,h,k)$ be the number of supernodes in a space-optimal HSST for $ST(N)$ under the restrictions: (a) the root of the HSST is an SST supernode for exactly k, $0 < k \leq K$, nodes of the binary trie $ST(N)$ (k is the *utilization* of the SST node) and (b) the height of the HSST is at most h. Let $D_t(N)$ be the descendents (in T) of N that are at level t of $ST(N)$.

There are two possibilities for the the root of $minHSST(ST(N),h)$, $h \geq 0$–the root is a TBM supernode or the root is an SST supernode. In the former case,

$$opt(N,h) = 1 + \sum_{R \in D_S(N)} opt(R, h-1) \tag{1}$$

and in the latter case,

$$opt(N,h) = \min_{0<k\leq K}\{opt(N,h,k)\} \tag{2}$$

Combining these two cases together, we get

$$opt(N,h) = \min\{1 + \sum_{R \in D_S(N)} opt(R,h-1), \min_{0<k\leq K}\{opt(N,h,k)\}\} \tag{3}$$

To simplify the recurrence for $opt(N,h,k)$, we introduce the function $f(N,h,k)$, which gives the number of supernodes in the space-optimal HSST for the binary trie composed of $ST(N)$ and the parent of N (we assume that N is not the root of T) under the restrictions: (a) the root of the HSST is an SST supernode whose utilization is $k+1$ and (b) the height of the HSST is at most h. Note that when $k=0$, the root of this HSST contains only the parent of N. So, $f(N,h,0) = 1 + opt(N,h-1)$. When $k>0$, the root represents a partition that includes the parent of N plus k nodes of $ST(N)$. So, $f(N,h,k) = opt(N,h,k)$. To obtain the recurrence for $opt(N,h,k)$, $h>0$ and $k>0$, we consider three cases–N has 0, 1, and 2 children.

When N has no child,

$$opt(N,h,k) = 1 \tag{4}$$

When N has only one child a,

$$opt(N,h,k) = f(a,h,k-1) \tag{5}$$

When N has two children a and b,

$$opt(N,h,k) = \min_{0\leq j<k}\{f(a,h,j) + f(b,h,k-j-1) - 1\} \tag{6}$$

Finally, for $h<0$, we have

$$opt(N,h,k) = opt(N,h) = \infty \tag{7}$$

and for $k \leq 0$, we have

$$opt(N,h,k) = \infty \tag{8}$$

as it isn't possible to represent $ST(N)$ by an HSST whose height is less than 0 or by an HSST whose root is an SST node with utilization ≤ 0.

Using (3), each $opt(*,*)$ value can be computed in $O(K)$ time, since $|D_S(N)| \leq 2^S \approx 2K$. Also, each $opt(*,*,*)$ value can be computed in $O(K)$ time using (4)-(8). There are $O(mH)$ $opt(*,*)$ and $O(mHK)$ $opt(*,*,*)$ values to compute. Hence, the time complexity is $O(mHK + mHK^2) = O(mHK^2) = O(nWHK^2)$, where n is the number of filters and W is the length of the longest prefix.

5 Implementation Considerations

5.1 HSSTs

If each supernode can be examined with a single memory access, then an HSST whose height is H (i.e., the number of levels is $H + 1$) may be searched for the next hop of the longest matching prefix by making at most $H + 2$ memory accesses. To get this performance, we must choose the supernode parameters K and S such that each type of supernode can be retrieved with a single access. As noted in Section 2, the size of a TBM node is $2^{S+1} + 2b - 1$ bits and that of an SST node is $4K + 2b - 1$ bits. An additional bit is needed for us to distinguish the two node types. So, any implementation of an HSST must allocate $2^{S+1} + 2b$ bits for a TBM node and $4K + 2b$ bits for an SST node. We refer to such an implementation as the *base implementation* of an HSST. Let B be the number of bits that may be retrieved with a single memory access and suppose that we use $b = 20$ bits for a pointer (as is done in [18]). When $B = 72$, our supernode parameters become $K = 8$ and $S = 4$. When $B = 64$, the supernode parameters become $K = 6$ and $S = 3$. Because of the need to align supernodes with word boundaries, each TBM node wastes 8 bits when $B = 64$.

Song et al. [18] have proposed an alternative implementation, called the *prefix-bit implementation*, for supernodes. This alternative implementation employs the prefix-bit optimization technique of Eatherton et al. [3]. An additional bit (called $prefixBit$) is added to each supernode. This bit is a 1 for a supernode N iff the search path through the parent supernode (if any) of N that leads us to N goes through a binary trie node that contains a prefix. With the $prefixBit$ added to each supernode, we may search an HSST as follows:

Step 1: Move down the HSST keeping track of the parent, Z, of the most recently seen supernode whose $prefixBit$ is 1. Do not examine the IBM of any node encountered in this step.

Step 2: Examine the IBM of the last supernode on the search path. If no matching prefix is found in this supernode, examine the IBM of supernode Z.

When prefix-bit optimization is employed, it is possible to have a larger K and S as the IBM (K or $2^S - 1$ bits) and NH (b bits) fields of a supernode are not accessed (except in Step 2). So, it is sufficient that the space needed by the remaining supernode fields be at most B bits. The IBM and NH fields may spill over into the next memory word. In other words, we select K and S to be the largest integers for which $3K + b + 1 \leq B$ and $2^S + b + 2 \leq B$. When $B = 72$ and $b = 20$, we use $K = 17$ and $S = 5$; and when $B = 64$ and $b = 20$, we use $K = 14$ and $S = 5$. When prefix-bit optimization scheme is employed, the number of memory accesses for a search is $H + 4$ as two additional accesses (relative to the base implementation) are needed to fetch the up to two IBMs and NH fields that may be needed in Step 2.

The additional access to the IBM of Z may be avoided by using *controlled leaf pushing*, which is quite similar to the standard leaf pushing proposed in [2]. Recall that each supernode of an HSST represents a subtree of the binary trie T for the forwarding table. In controlled leaf pushing, we examine the root N of the binary subtree represented by each supernode. If N contains no next hop, we add to N the next hop of the longest prefix that matches $Q(N)$. Note that when controlled leaf pushing is used, no $prefixBit$ is needed and we do not need to keep track of the parent node Z during a lookup. We refer to this implementation of HSSTs with controlled leaf pushing as the *enhanced* prefix-bit implementation. The number of memory accesses required for a lookup in an enhanced prefix-bit implementation is $H + 3$.

5.2 Base Implementation Optimization

When the base implementation is used and $b = 20$, we can increase the value of K by 5 if we eliminate the NH pointer (for a saving of b bits). The elimination of the NH pointer may also lead to an increase in S. To eliminate the NH pointer, we store the next-hop array, NA, of a supernode N next to its child array, CA. The start of the next-hop array for N can be computed from the child pointer of N and knowledge of the number of children supernodes that N has. The latter may be determined from the EBM of N. Since the size of a next-hop array may not be a multiple of B, this strategy may result in each next-hop array wasting up to $B - 1$ bits as each child array must be aligned at a word boundary. We can reduce the total number of words of memory used by this *enhanced* base implementation if we pair some of the (CA, NA) pairs and flip the second (CA, NA) tuple in each pair. For example, suppose that $B = 72$, each next-hop entry uses 18 bits, $NA1$ requires 162 bits, and $NA2$ requires 180 bits. Each entry in a child array is a supernode that uses B bits. Since each (CA, NA) must start at a word boundary, placing $(CA1, NA1)$ and $(CA2, NA2)$ into memory uses $n_1 + n_2 + 6$ B-bit words, where n_1 and n_2 are, respectively, the number of supernodes in $CA1$ and $CA2$. If we flip $(CA2, NA2)$ to get $(NA2, CA2)$ then the next-hop array $NA2$ can use 36 of the 54 bits of a B-bit word not used by $NA1$ and reduce the total word count by 1. This sharing of a B-bit word by $NA1$ and $NA2$ leaves 18 unused bits in the shared B-bit word and the child array $CA2$ remains aligned to a word boundary. The child pointer for $(NA2, CA2)$ now points to the start of the array $NA2$ and to compute the start of the array $CA2$ from this child pointer, we must know the number of next-hop entries in $NH2$. This number can be determined from the IBM. To employ this flipping strategy to potentially reduce the total memory required by the enhanced base implementation, each supernode must be augmented with a bit that identifies the orientation (CA, NA) or (NA, CA) used for its child and next-hop arrays.

To minimize the memory used by the enhanced base implementation, we must solve the following restricted bin packing problem (RBIN): pack n integers $b_1, \cdots, b_n$ in the range $[1, B)$, into the smallest number of size B buckets such that no bucket is assigned more than two of the integers. The RBIN problem may be solved in $O(n \log n)$ time by using the first-fit decreasing heuristic modified so as to pack at most two items in each bin. The optimality of this strategy is easily established by induction on n. An alternative strategy is to sort the b_is into decreasing order and then to repeatedly pair the smallest unpaired b_i with the largest unpaired b_i (under the constraint that the sum of the paired b_is is no more than B). The pairing process terminates when no new pair can be created. The number of remaining singletons and pairs is the minimum number of bins needed for the packing.

5.3 End-Node Optimized HSSTs

A further reduction in the space requirements of an HSST may be achieved by employing end-node optimization [3]. We permit four formats for a leaf supernode. Fig. 6 shows these four formats for the base implementation. Each supernode (leaf or non-leaf) uses a bit to distinguish between leaf and non-leaf supernodes. Each leaf supernode uses two additional bits to distinguish among the four leaf formats while each non-leaf supernode uses an additional bit to distinguish between SST and TBM supernodes.

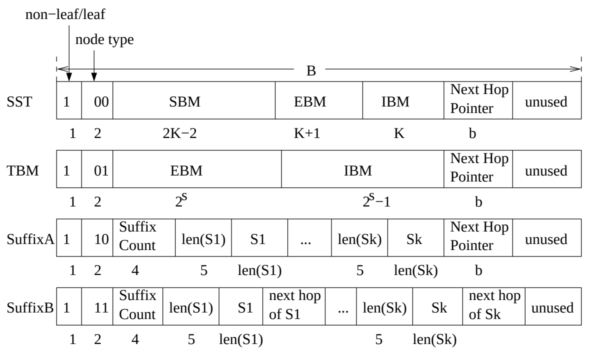


Figure 6: Leaf supernode formats

The leaf supernodes are obtained by identifying the largest subtries of the binary trie T that fit into one of the four leaf-supernode formats. Notice that a leaf supernode has no child pointer. Consequently, in the SST format we may use a larger K than used for non-leaf supernodes and in the TBM format, a larger S may be possible. The third format (SuffixA) is used when we may pack the prefixes in a subtrie into a single supernode. For this packing, let N be the root of the subtrie being packed. Then, $Q(N)$ (the prefix defined by the path from the root of T to N) is the same for all prefixes in the subtrie rooted at N. Hence the leaf supernode need store only the suffixes obtained by deleting $Q(N)$ from each prefix in $ST(N)$. The leaf supernode stores the number of these suffixes, followed by pairs of the form (suffix length, suffix). In Fig. 6, $len(S1)$ is the length of the first suffix and $S1$ is the first suffix in the supernode. Leaf supernodes in the third format are searched by serially examining the suffixes stored in the node and comparing these with the destination address (after this is stripped of the prefix $Q(N)$; this stripping may be done as we move from $root(T)$ to N). The fourth format (SuffixB), which is similar to the third format, avoids the memory access required by the third format to extract the next hop. When controlled leaf pushing is applied to SuffixB supernodes, the worst-case number of memory accesses required for a lookup may decrease. Note that in the absence of controlled leaf pushing, if no matching prefix is found in a SuffixB leaf supernode, we would need an additional access to extract the next hop associated with the longest matching prefix along the search path.

For all $ST(N)$s may be represented by a leaf supernode of the first three types, we set $opt(N, h) = 1$ for $h \geq 0$ and for all $ST(N)$s that may be represented by a SuffixB supernode, we set $opt(N, h) = 1$ for $h \geq -1$. The dynamic programming recurrence of Section 4 is then used to determine $opt(root(T), H)$.

Although we have described end-node optimization only for the base implementation, this technique may be applied to the enhanced prefix-bit implementation as well to reduce total memory requirement.

6 Experimental Results

C++ codes for our algorithms for space-optimal supernode tries were compiled using the GCC 3.3.5 compiler with optimization level `O3` and run on a 2.80 GHz Pentium 4 PC. Our algorithms were benchmarked against recently published algorithms to construct space-efficient data structures for packet forwarding [18, 19]. The benchmarked algorithms seek to construct lookup structures that (a) minimize the worst-case number of memory accesses needed for a lookup and (b) minimize the total memory needed to store the constructed data structure. As a result, our experiments measured only these two quantities. Further, all test algorithms were run so as to generate a lookup structure that minimizes the worst-case number of memory accesses needed for a lookup; the size (i.e., memory required) of the constructed lookup structure was minimized subject to this former constraint. For benchmarking purposes we assumed that the forwarding table data structure will reside on a QDRII SRAM, which supports both B=72 bits (dual burst) and B=144 bits (quad burst). For our experiments, we used $b = 22$ bits for a pointer (whether a child pointer or a pointer to a next-hop array) and 12 bits for each next hop.

Four variants of our space-optimal HSST were implemented–*enhanced prefix-bit* (EP), *enhanced prefix-bit with end-node optimization* (EPO), *enhanced base* (EB), and *enhanced base with end-node optimization* (EBO). In addition, we considered the BFP algorithm of Song et al. [18][2] and the variant 3 algorithm (which we refer to as V3MT [3]) of [19] to construct multi-way trees. Extensive experiments reported in [19] establish the superiority of V3MT, in terms of space and lookup efficiency, over other known schemes for space and time efficient representation of IP lookup tables. [18] establishes the superiority of BFP over TBM [3]. However, [18] did not compare BFP to V3MT.

First, we report on the IPv4 experiments, which were conducted using the six IPv4 router tables Aads, MaeWest, RRC01, RRC04, AS4637 and AS1221 that were obtained from [13, 14, 5]. The number of prefixes in these router tables is 17486, 29608, 103555, 109600, 173501 and 215487, respectively.

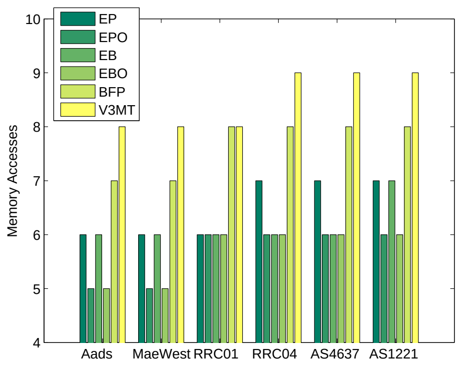
Fig. 7 plots the number of memory accesses required for a lookup in the data structure constructed by each of our algorithms (assuming the root is held in a register). Unlike the access counts reported in [18, 19], the numbers reported by us include the additional access (if any) needed to obtain the next hop for the longest matching prefix. As can be seen, EBO results in the smallest access counts for all of our test sets; EPO ties with EBO on all of the six test sets when $B = 72$ (our other experiments with 9-bit next hop and 18-bit pointer fields indicate that EBO often requires one memory access less than EPO when B=72) and on 2 of the test cases when $B = 144$. The number of memory accesses for a lookup in the structure constructed by BFP ranges from 1.33 to 2.00 times that required by the EBO structure; on average the BFP structure requires 1.53 times the number of accesses required by the EBO structure and the standard deviation is 0.25. The number of memory accesses required by V3MT

[2]The BFP algorithm of [18] is flawed as during its generation of TBM nodes it doesn't check if part of the underlying full tree has already been pruned to construct another supernode. Our implementation fixes this flaw using a fix discussed with the authors of [18].

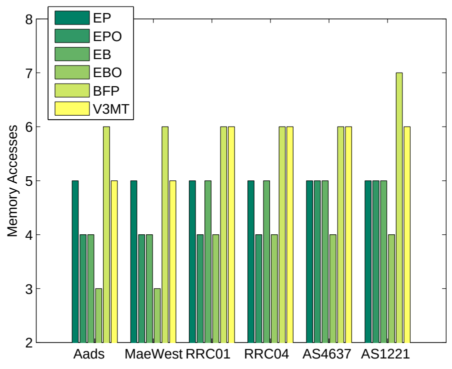
[3]We are grateful to the authors of [19] for providing their code.

structure normalized by that required by EBO structure is between 1.33 and 1.67 (the mean and the standard deviation are 1.53 and 0.09).

The number of memory accesses required by the structures constructed by each of our 6 test algorithms reduces when B goes from $B = 72$ to $B = 144$. The reduction for EPO is between 17% and 33% (the mean and standard deviation are 23% and 8%). The reduction for EBO is from 33% to 40% (the mean and standard deviation are 36% and 3%). Notice that when $B = 72$, BFP outperformed V3MT by 1 memory access on 5 of the 6 data sets and tied on the sixth. However, when $B = 144$, V3MT outperformed BFP by 1 memory access on 3 of the 6 data sets and tied on the remaining 3.

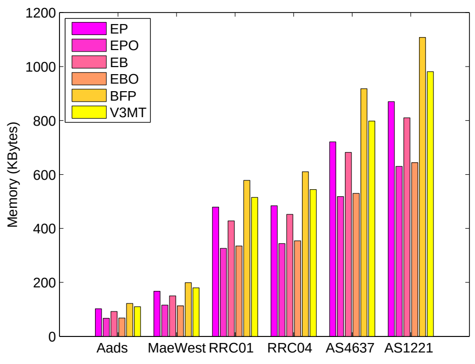


(a) B=72bits

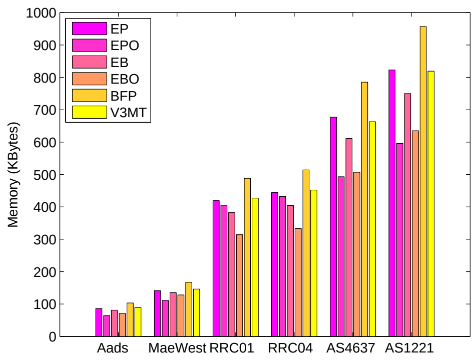


(b) B=144bits

Figure 7: Number of memory accesses required for lookup in IPv4 tables



(a) B=72bits



(b) B=144bits

Figure 8: Total memory (KBytes) required for IPv4 tables

Fig. 8 plots the total memory required by the lookup structure constructed by each of our 6 algorithms. As can be seen, EPO and EBO result in the least total memory requirement. Although EPO is slightly superior to EBO on the memory measure on 9 of our 12 test cases, the total memory required by EBO for all 12 test cases is 2% less than that required by EPO. The search structures constructed by the remaining algorithms required, on average, between 23% and 61% more memory than did the structures constructed by EBO. When $B = 72$, the average number of bits of storage needed per prefix is 48 for for BFP, 42 for V3MT and 27 for EBO. The corresponding numbers for the case when $B = 144$ are 41, 35, and 27.

When B is increased from 72 to 144, the memory required by EPO and EBO decreased for 4 of the 6 data sets and increased for the remaining 2. The $B = 144$ memory normalized by the $B = 72$ memory is between 0.95 and 1.26, the average and standard deviation being 1.05 and 0.15, respectively. For EBO, the corresponding normalized numbers were 0.96, 1.13, 1.0, and 0.07.

On our IPv4 data sets, EBO and EPO are the clear winners. EBO is slightly superior to EPO on the memory access measure and the two are very competitive on the memory required measure. Since the former is more important, we recommed EBO over EPO. The EBO lookup structures require 25% to 50% fewer accesses than do the BFP structures; they also reduce memory requirement by 24% to 44%. The reduction in number of memory accesses and memory requirement relative to V3MT are 25% to 40% and 12% to 38%.

In [8], we report the observed superiority of space-optimal HSSTs over other succinct structures [2, 9, 10] and improvement on IPv6 router tables as well.

7 Conclusion

We have developed a fast algorithm to construct minimum-height SSTs. Our algorithm reduces the complexity of this construction from $O(m^2)$ [18] to $O(m)$, where m is the number of nodes in the input binary trie. Additionally, we have developed dynamic programming formulations for the construction of space-optimal HSSTs. Our experiments indicate that for IPv4 data sets, our EBO structures require between 25% and 50% fewer memory accesses for a lookup than required by the HSST structure of [18]. Additionally, our EBO structures require between 24% and 44% less memory. Compared to the structures produced by the V3MT algorithm of [19], our EBO structures require between 25% and 40% fewer memory accesses and between 12% and 38% less memory.

References

[1] IPv6 Address Allocation and Assignment Policy (APNIC), *http://www.apnic.net/docs/policy/ipv6-address-policy.html*

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

[3] W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software IP lookups with incremental updates, *Computer Communication Review*, 34(2): 97-122, 2004.

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

[5] *http://www.merit.edu/ipma/routing_table*

[6] G. Jacobson, Succinct Static Data Structure, *Carnegie Mellon University Ph.D Thesis*, 1998.

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

[8] W. Lu and S. Sahni, Succinct Representation of Packet Classifiers, University of Florida, 2006, *http://www.cise.ufl.edu/∼wlu/papers/hsst.pdf.*

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

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

[11] J. Munro and S. Rao, Succinct representation of data structures, in *Handbook of Data Structures and Applications*, D. Mehta and S. Sahni editors, Chapman & Hall/CRC, 2005.

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

[13] *http://bgp.potaroo.net*

[14] Ris, Routing information service raw data, *http://data.ris.ripe.net*

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

[16] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Journal on Foundations of Computer Science*, 14, 3, 2003, 337-358.

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

[18] H. Song, J. Turner, and J. Lockwood, Shape Shifting Tries for Faster IP Route Lookup, *Proceedings of 13th IEEE International Conference on Network Protocols*, 2005.

[19] X.Sun and Y.Zhao, An On-Chip IP Address Lookup Algorithm, *IEEE Transactions on Computers*, 2005, 873-885.

[20] M. Wang, S. Deering, T. Hain, and L. Dunn, Non-random Generator for IPv6 Tables, *12th Annual IEEE Symposium on High Performance Interconnects*, 2004.