Succinct Representation Of Static Packet Classifiers *

Wencheng Lu and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
{wlu, sahni}@cise.ufl.edu
July 31, 2006

Abstract
We develop algorithms for the compact representation of the 1- and 2-dimensional tries that are used for Internet packet classification. Our compact representations are experimentally compared with competing compact representations for 1- and 2-dimensional packet classifiers and found to simultaneously reduce the number of memory accesses required for a lookup as well as the memory required to store the classifier.

Keywords
Packet classification, succinct representation, 1- and 2-dimensional tries, dynamic programming.

1 Introduction
An Internet router classifies incoming packets based on their header fields using a classifier, which is a table of rules. Each classifier rule is a pair \((F, A)\), where \(F\) is a filter and \(A\) is an action. If an incoming packet matches a filter in the classifier, the associated action specifies what is to be done with this packet. Typical actions include packet forwarding and dropping. A \(d\)-dimensional filter \(F\) is a \(d\)-tuple \((F[1], F[2], \cdots, F[d])\), where \(F[i]\) is a range that specifies destination addresses, source addresses, port numbers, protocol types, TCP flags, etc. A packet is said to match filter \(F\), if its header field values fall in the ranges \(F[1], \cdots, F[d]\). Since it is possible for a packet to match more than one of the filters in a classifier, a tie breaker is used to determine a unique matching filter.

In one-dimensional packet classification (i.e., \(d = 1\)), \(F[1]\) is usually specified as a destination address prefix and lookup involves finding the longest prefix that matches the packet’s destination address. Data structures for longest-prefix matching have been extensively studied (see [33, 34], for surveys). Although 1-dimensional prefix filters are adequate for destination based packet forwarding, higher dimensional filters are required for firewall, quality of service, and virtual private network applications, for example. Two-dimensional prefix filters, for example, may be used “to represent host to host or network to network or IP multicast flows” [11] and higher dimensional filters are required if these flows are to be represented “with greater granularity.” Eppstein and Muthukrishnan [8] state that “Some proposals are underway to specify many fields ... while others are underway which seem to preclude using more than just the source and destination IP addresses ... (in IPsec for example, the source or destination port numbers may not be revealed).” Kaufman et al. [15] also point out that in IPsec, for security reasons, fields other

---

*This research was supported, in part, by the National Science Foundation under grant ITR-0326155
than the source and destination address may not be available to a classifier. *Thus two-dimensional prefix filters represent an important special case of multi-dimensional packet classification.* Data structures for multi-dimensional (i.e., \(d > 1\)) packet classification are developed in [2, 3, 4, 5, 8, 9, 10, 11, 16, 31, 37, 39, 36, 19, 20, 22, 23, 43], for example.

Our focus in this paper is the succinct representation of the 1- and 2-dimensional tries that are commonly used to represent 1- and 2-dimensional classifiers; the succinct representation is to support high-speed packet classification. For this work, we assume that the classifier is static. That is, the set of rules that comprise the classifier does not change (no inserts/deletes). This assumption is consistent with that made in most of the classifier literature where the objective is to develop a memory-efficient classifier representation that can be searched very fast.

We begin, in Section 2, by reviewing the 1- and 2-dimensional binary trie representation of a classifier together with the research that has been done on the succinct representation of these structures. In Sections 3 through 7.3, we develop our our algorithms for the succinct representation of 1-dimensional tries and in Sections 5 and 6, we do this for 2-dimensional tries. Experimental results are presented in Section 8.

## 2 Background and Related Work

### 2.1 One-Dimensional Packet Classification

We assume that the filters in a 1-dimensional classifier are prefixes of destination addresses. Many of the data structures developed for the representation of a 1-dimensional classifier are based on the *binary trie* structure [12].

A binary trie is a binary tree structure in which each node has a data field and two children fields. 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 \(i\)th 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.

Figure 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. Figure 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 Figure 1 (b) differs from the 1-bit trie used in [38], [34], 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).

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 [29], Lulea [6], tree bitmap [7], multibit tries [38], shape shifting tries [40]—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 [26, 27] 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. [40], which in turn draws heavily from the tree bitmap (TBM) scheme of Eatherton et al. [7] and the technique developed by Jacobson [14, 28] for the succinct representation of a binary tree. In TBM we start with the binary trie for our classifier and partition this binary trie into subtrees 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 [7] for real-world IPv4 classifiers, we use \( S = 2 \) here to illustrate the TBM structure.

Figure 2 (a) shows a partitioning of the binary trie of Figure 1 (b) into 4 subtrees 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 represented by a supernode (Figure 2 (b)) that has the following components:

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. [40] is obtained by partitioning a binary trie into subtrees 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. [40] is the succinct representation of a binary tree developed by Jacobson [14]. 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 Figure 1 (b) into binary tries that have at most \(K = 3\) nodes each. Figure 3 (a) shows a possible partitioning into the 3 partitions X-Z. X includes nodes a, b and d of Figure 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. Figure 3 (b) shows the node representation for each partition of Figure 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 output 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 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. [40] 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. [40] 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. Figure 4 shows an HSST for the binary trie of Figure 1 (b). For this HSST, $K = S = 2$. The HSST has two SST nodes X and Z, and one TBM node Y.

Although Song et al. [40] 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.

### 2.2 Two-Dimensional Packet Classification

We assume that the filters are of the form $(D, E)$, where $D$ is a destination address prefix and $E$ is a source address prefix. A 2-dimensional classifier may be represented as a 2-dimensional binary trie (2DBT), which is a one-dimensional binary trie (called the top-level trie) in which the data field of each node is a pointer to a (possibly empty) binary trie (called the lower-level trie). So, a 2DBT has 1 top-level trie and potentially many lower-level

![Figure 3: SST for binary trie of Figure 1 (b)](image)
tries.

Consider the 5-rule two-dimensional classifier of Figure 5. For each rule, the filter is defined by the Dest (destination) and Source prefixes. So, for example, \( F_2 = (0^*,1^*) \) matches all packets whose destination address begins with 0 and whose source address begins with 1. When a packet is matched by two or more filters, the matching rule with least cost is used. The classifier of Figure 5 may be represented as a 2DBT in which the top-level trie is constructed using the destination prefixes. In the context of our destination-source filters, this top-level trie is called the destination trie (or simply, dest trie). Let \( N \) be a node in the destination trie. If no dest prefix equals \( Q(N) \), then \( N.data \) points to an empty lower-level trie. If there is a dest prefix \( D \) that equals \( Q(N) \), then \( N.data \) points to a binary trie for all source prefixes \( E \) such that \( (D,E) \) is a filter. In the context of destination-source filters, the lower-level tries are called source trees.

Every node \( N \) of the dest trie of a 2DBT has a (possibly empty) source trie hanging from it. Let \( a \) and \( b \) be two nodes in the dest trie. Let \( b \) be an ancestor of \( a \). We say that the source trie that hangs from \( b \) is an ancestor trie of the one that hangs from \( a \). Figure 6 gives the 2DBT for the filters of Figure 5.
Srinivasan and Varghese [36] proposed using two-dimensional one-bit tries, a close relative of 2DBTs, for destination-source prefix filters. The proposed two-dimensional trie structure takes $O(nW)$ memory, where $n$ is the number of filters in the classifier and $W$ is the length of the longest prefix. Using this structure, a packet may be classified with $O(W^2)$ memory accesses. The basic two-dimensional one-bit trie may be improved upon by using pre-computation and switch pointers [36]. The improved version classifies a packet making only $O(W)$ memory accesses. Srinivasan and Varghese [36] also propose extensions to higher-dimensional one-bit tries that may be used with $d$-dimensional, $d > 2$, filters. Baboescu et al. [4] suggest the use of two-dimensional one-bit tries with buckets for $d$-dimensional, $d > 2$, classifiers. Basically, the destination and source fields of the filters are used to construct a two-dimensional one-bit trie. Filters that have the same destination and source fields are considered to be equivalent. Equivalent filters are stored in a bucket that may be searched serially. Baboescu et al. [4] report that this scheme is expected to work well in practice because the bucket size tends to be small. They note also that switch pointers may not be used in conjunction with the bucketing scheme.

Lu and Sahni [22], develop fast polynomial-time algorithms to construct space-optimal constrained 2DMTs (two-dimensional multibit tries). The constructed 2DMTs may be searched with at most $k$ memory accesses, where $k$ is a design parameter. The space-optimal constrained 2DMTs of Lu and Sahni [22] may be used for $d$-dimensional filters, $d > 2$, using the bucketing strategy proposed in [4]. For the case $d = 2$, switch pointers may be employed to get multibit tries that require less memory than required by space-optimal constrained 2DMTs and that permit packet classification with at most $k$ memory accesses. Lu and Sahni [22] develop also a fast heuristic to construct good multibit tries with switch pointers. Experiments reported in [22] indicate that, given the same memory budget, space-optimal constrained 2DMT structures perform packet classification using $1/4$ to $1/3$ as many memory accesses as required by the two-dimensional one-bit tries of [36, 4].
3 Minimum-Height SSTs

The breadth-first pruning algorithm of Song et al. [40] 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 \( \text{minHtSST} \), 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 \( \text{minHtSST} \) 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 \( \text{SSTs that hang from } x \). 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 \( \text{left hanging} \) SSTs; the remaining SSTs are the \( \text{right hanging} \) SSTs of \( x \). We use the following notation: \( x.\text{leftChild} \) (\( x.\text{rightChild} \)) is the left (right) child of \( x \) in \( T \); \( x.\text{st} \) is the set of nodes in the subtrie of \( T \) rooted at \( x \); \( x.\text{rn} \) is the subset of \( x.\text{st} \) that have not been pruned off at the time \( x \) is visited; \( x.\text{size} \) is the number of nodes in \( x.\text{rn} \); \( x.\text{SSTs} \) is the set of SSTs that hang from \( x \) at the time \( x \) is visited; \( x.\text{leftSSTs} \) (\( x.\text{rightSSTs} \)) is the subset of \( x.\text{SSTs} \) that are left (right) hanging SSTs. \( x.lht = -1 \) (left height) if \( x.\text{leftSSTs} \) is empty. Otherwise, \( x.lht \) is the maximum height of an SST in \( x.\text{leftSSTs} \) (the height of an SST is 1 less than the number of levels in the tree). \( x.rht \) is the corresponding quantity for the \( x.\text{rightSSTs} \) and \( x.\text{ht max} \{x.lht, x.rht\} \).

The function \( \text{prune}(y) \) prunes \( T \) at the node \( y \) by removing all nodes in \( y.\text{rn} \). The nodes in \( y.\text{rn} \) are used to create a supernode whose subtries are \( y.\text{SSTs} \). When \( y \) is NULL, \( \text{prune}(y) \) is a NULL operation. Figure 7 gives the visit function employed by our postorder traversal algorithm \( \text{minHtSST} \). \( x \) is the node of \( T \) being visited. To avoid code clutter, we do not show the code needed to update \( \text{size}, \text{lht}, \text{rht}, \text{SSTs}, \) 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 Figure 7, then \( x.\text{leftChild}.\text{size} < K \) and \( x.\text{rightChild}.\text{size} < K \) 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 \( x.\text{rn} \) has \( y.\text{ht} = x.lht \) (\( x.rht \)).

**Lemma 1** When \( x \) is visited, every node \( y \) that is in the left (right) subtree of \( x \) and in \( x.\text{rn} \) has \( y.\text{ht} = x.lht \) (\( x.rht \)).

**Proof** Let \( u \) be the nearest left descendent (i.e., descendent in the left subtrie) of \( x \) that is in \( x.\text{rn} \) and has \( u.\text{ht} \neq x.lht \). If there is no such \( u \), then \( y.\text{ht} = x.lht \) for every left descendent \( y \) of \( x \) that is in \( x.\text{rn} \). So, assume there is such a \( u \). Clearly, \( u.\text{ht} < x.lht \). \( u \) cannot be the left child of \( x \) as otherwise \( x.\text{leftSSTs} = u.\text{SSTs} \) and so \( x.lht = u.\text{ht} \). So, \( u \) has a parent \( v \) that is in \( x.\text{rn} \) (\( v \) also is a left descendent of \( x \)) and \( u.\text{ht} < v.\text{ht} = x.lht \). Without loss of generality, assume that \( u \) is the left child of \( v \). So, \( v.\text{ht} = u.\text{ht} < v.\text{ht} = v.rht \). During our postorder traversal algorithm is quite different from the postorder traversal algorithm \( \text{POP} \) of [40], which constructs an SST that has the fewest number of nodes.

---

\(^1\)Our postorder traversal algorithm is quite different from the postorder traversal algorithm \( \text{POP} \) of [40], 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 7: Visit function for minHtSST

traversal of \(T\), node \(v\) is visited before node \(x\). When \(v\) was visited, \(v\)'s left subtree (including node \(u\)) should have been pruned from \(T\) (Case 2 of the visit function) and so \(u\) cannot be in \(x.rn\), a contradiction.

The proof for nodes that are in the right subtree of \(x\) and in \(x.rn\) is similar.

\[ \text{Theorem 1} \]

For every binary trie \(T\) and integer \(K > 0\), the postorder traversal algorithm minHtSST constructs an SST that has minimum height.

\[ \text{Proof} \]

We establish this result by showing that if there is a minimum height SST for \(T\) that includes all the supernodes constructed up to (but not including) the time a node \(x\) is visited, then there is a minimum height SST for \(T\) that includes all the supernodes constructed up to and including the visit of \(x\). Since the antecedent of this statement is true when we visit the first node of \(T\), the theorem follows by induction.

Consider the visit of any node \(x\) of \(T\). Let \(U\) be an SST that includes all supernodes constructed by the algorithm up to this time. If, during the visit of \(x\), no new supernodes are constructed, then we have nothing to prove. So, assume that one or more supernodes are constructed. Let \(P\) be the (super)node of \(U\) that includes \(x\).

Suppose the new supernodes are constructed in Case 1 of the visit function. So, \(x.lht = x.rht\) and \(x.size \geq K\).
If \(x.size > K\), \(P\) cannot include all nodes of \(x.rn\). So, some of the nodes of \(x.rn\) are contained in descendents of \(P\). From Lemma 1, we get \(y.ht = x.lht = x.rht = x.ht\) for every \(y \in x.rn\) other than \(y = x\). Hence, the height of the subSST of \(U\) rooted at \(P\) is at least \(x.ht + 2\). Now, delete the nodes of \(x.rn\) (other than \(x\)) from the supernodes of \(U\), delete from \(U\) any supernodes that become empty, and add to \(U\) the two supernodes constructed by \(\text{prune}(x.leftChild)\) and \(\text{prune}(x.rightChild)\). We get an SST \(U'\) for \(T\) whose height is no more than that of \(U\) and which contains all the supernodes constructed up to and including the visit of \(x\). If \(x.size = K\), \(P\) may include all nodes of \(x.rn\). In this case, \(U\) has all the supernodes constructed by our algorithm up to and including the visit of \(x\). If \(P\) does not include all nodes of \(x.rn\), some nodes of \(x.rn\) must be in descendent nodes of \(P\) and so, as before, the height of the subSST rooted at \(P\) is at least \(x.ht + 2\). Now, delete the nodes of \(x.rn\) from the supernodes of \(U\), delete from \(U\) any supernodes that become empty, and add to \(U\) the supernode constructed by
prune(x). We get an SST $U'$ for T whose height is no more than that of $U$ and which contains all the supernodes constructed up to and including the visit of $x$.

If the new supernode is constructed in Case 2, the height of the subSST of $T$ rooted at $P$ is at least $x.rht + 1$. Delete the left descendents of $x$ that are in $x.rn$ from the supernodes of $U$, delete from $U$ any supernodes that become empty, and add to $U$ the supernode constructed by prune($x$.leftChild). We get an SST $U'$ for $T$ whose height is no more than that of $U$. Now, if $x$.size = $K$, we do the transformation given above in Case 1 ($x$.size = $K$) and obtain an SST whose height is no more than that of $U$ and which contains all the supernodes constructed up to and including the visit of $x$.

The proof for Case 3 is similar to that for Case 2.

Since the visit function of Figure 7 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 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,

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

and in the latter case,

$$\text{opt}(N, h) = \min_{0 < k \leq K} \{ \text{opt}(N, h, k) \}$$ \hspace{1cm} (2)

Combining these two cases together, we get

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

To simplify the recurrence for $\text{opt}(N, h, k)$, 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, $f(N, h, 0) = 1 + \text{opt}(N, h - 1)$. When $k > 0$, the root represents a partition that includes the parent of $N$ plus $k$ nodes of $ST(N)$. So, $f(N, h, k) = \text{opt}(N, h, k)$. To obtain the recurrence for $\text{opt}(N, h, k)$, $h > 0$ and $k > 0$, we consider three cases—$N$ has 0, 1, and 2 children.

When $N$ has no child,

$$\text{opt}(N, h, k) = 1$$ \hspace{1cm} (4)

When $N$ has only one child $a$,

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

When $N$ has two children $a$ and $b$,

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

Finally, for $h < 0$, we have

$$\text{opt}(N, h, k) = \text{opt}(N, h) = \infty$$ \hspace{1cm} (7)

and for $k \leq 0$, we have

$$\text{opt}(N, h, k) = \infty$$ \hspace{1cm} (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 $\leq 0$.

Using Equation 3, each $\text{opt}(\ast, \ast)$ value can be computed in $O(K)$ time, since $|D_S(N)| \leq 2^S \approx 2K$. Also, each $\text{opt}(\ast, \ast, \ast)$ value can be computed in $O(K)$ time using Equations 4-8. There are $O(mH) \text{opt}(\ast, \ast)$ and $O(mHK) \text{opt}(\ast, \ast, \ast)$ values to compute. Hence, the time complexity is $O(mHK + mHK^2) = O(mHK^2)$ $O(nW HK^2)$, where $n$ is the number of filters and $W$ is the length of the longest prefix.
Figure 8: Two-dimensional supernode trie for Figure 5

5 Space-Optimal 2DHSSTs

Let $T$ be a 2DBT. We assume that the source tries of $T$ have been modified so that the last prefix encountered on each search path is the least-cost prefix for that search path. This modification is accomplished by examining each source-trie node $N$ that contains a prefix and replacing the contained prefix with the least-cost prefix on the path from the root to $N$. A 2DHSST may be constructed from $T$ by partitioning the top-level binary trie (i.e., the dest trie) of $T$ and each lower-level binary trie into a mix of TBM and SST supernodes. Supernodes that cover the top-level binary trie use their $NH$ (next hop) lists to store the root supernodes for the lower-level HSSTs that represent lower-level tries of $T$.

Figure 8 shows a possible 2DHSST for the 2DBT of Figure 6. The supernode strides used are $K = S = 2$. A 2DHSST may be searched for the least-cost filter that matches any given pair of destination and source addresses $(da, sa)$ by following the search path for $da$ in the destination HSST of the 2DHSST. All source tries encountered on this path are searched for $sa$. The least-cost filter on these source-trie search paths that matches $sa$ is returned. Suppose we are to find the least-cost filter that matches $(000,111)$. The search path for 000 takes us first to the root (ab) of the 2DHSST of Figure 8 and then to the left child (dg). In the 2DHSST root, we go through nodes a and b of the dest binary trie and in the supernode dg, we go through nodes d and g of $T$. Three of the encountered nodes (a, b, and g) have a hanging source trie. The corresponding source HSSTs are searched for 111 and F2 is returned as the least-cost matching filter.

To determine the number of memory accesses required by a search of a 2DHSST, we assume sufficient memory bandwidth that an entire supernode (this includes the IBM, EBM, child and $NH$ pointers) may be accessed with
a single memory reference. To access a component of the NH array, an additional memory access is required. For each supernode on the search path for da, we make one memory access to get the supernode’s fields (e.g., IBM, EBM, child and NH pointers). In addition, for each supernode on this path, we need to examine some number of hanging source HSSTs. For each source HSST examined, we first access a component of the dest-trie supernode’s NH array to get the root of the hanging source HSST. Then we search this hanging source HSST by accessing the remaining nodes on the search path (as determined by the source address) for this HSST. Finally, the NH component corresponding to the last node on this search path is accessed. So, in the case of our above example, we make 2 memory accesses to fetch the 2 supernodes on the dest HSST path. In addition, 3 source HSSTs are searched. Each requires us to access its root supernode plus an NH component. in each source HSST. The total number of memory accesses is $2 + 2 * 3 = 8$.

Let $MNMA(X)$ be the maximum number of memory accesses (MNMA) required to search a source HSST $X$. For a source HSST, the MNMA includes the access to NH component of the last node on the search path. So, $MNMA(X)$ is one more than the number of levels in $X$. Let $U$ be a 2DHSST for $T$ with strides $S$ and $K$. Let $P$ be any root to leaf path in the top level HSST of $U$. Let the sum of the MNMAs for the lower-level HSSTs on the path $P$ be $H(P)$. Let $nodes(P)$ be the number of supernodes on the path $P$. Define $2DHSST(h)$ to be the subset of the possible 2DHSSTs for $T$ for which

$$\max_P \{H(P) + nodes(P)\} \leq h$$  \hspace{1cm} (9)

Note that every $U, U \in 2DHSST(h)$, can be searched with at most $h$ memory accesses per lookup. Note also that some 2DHSSTs that have a path $P$ for which $H(P) + nodes(P) = h$ can be searched with fewer memory accesses than $h$ as there may be no (da, sa) that causes a search to take the longest path through every source HSST on paths $P$ for which $H(P) + nodes(P) = h$.

We consider the construction of a space-optimal 2DHSST $V$ such that $V \in 2DHSST(H)$. We refer to such a $V$ as a space-optimal 2DHSST($h$). Let $N$ be a node in $T$’s top-level trie, and let $2DBT(N)$ be the 2-dimensional binary trie rooted at $N$. Let opt1($N, h$) be the size (i.e., number of supernodes) of the space-optimal 2DHSST($h$) for $2DBT(N)$. opt1(root($T$), $H$) gives the size of a space-optimal 2DHSST($H$) for $T$. Let $g(N, q, h)$ be the size (excluding the root) of a space-optimal 2DHSST($h$) for $2DBT(N)$ under the constraint that the root of the 2DHSST is a TBM supernode whose stride is $q$. So, $g(N, S, h) + 1$ gives the size of a space-optimal 2DHSST($h$) for $2DBT(N)$ under the constraint that the root of the 2DHSST is a TBM supernode whose stride is $S$. We see that, for $q > 0$,

$$g(N, q, h) = \min_{m(N) \leq i \leq h} \{g(LC(N), q - 1, h - i) + g(RC(N), q - 1, h - i) + s(N, i)\}$$  \hspace{1cm} (10)

where $m(N)$ is the minimum possible value of MNMA for the source trie (if any) that hangs from the node $N$ (in case there is no source trie hanging from $N$, $m(N) = 0$), $g(N, 0, h) = opt1(N, h - 1)$, $g(null, t, h) = 0$, and $LC(N)$ and $RC(N)$ respectively, are the left and right children (in $T$) of $N$. $s(N, i)$ is the size of the space-optimal HSST for the source trie that hangs off from $N$ under the constraint that the HSST has an MNMA of at most $i$. $s(N, i)$
is 0 if $N$ has no hanging source trie.

Let $opt_1(N, h, k)$ be the size of a space-optimal $2DHSST(h)$ for $2DBT(N)$ under the constraint that the root of the $2DHSST$ is an SST supernode whose utilization is $k$. It is easy to see

$$opt_1(N, h) = \min \{ g(N, S, h) + 1, \min_{0 \leq k \leq K} \{ opt_1(N, h, k) \} \} \tag{11}$$

Suppose that $k > 0$ and $h > 0$. If $N$ has no child,

$$opt_1(N, h, k) = 1 + s(N, h - 1) \tag{12}$$

When $N$ has only one child $a$,

$$opt_1(N, h, k) = \min_{m(N) \leq i < h} \{ f(a, h - i, k - 1) + s(N, i) \} \tag{13}$$

where $f(N, h, k)$ is the size of a space-optimal $2DHSST(h)$ for $2DBT(N)$ plus the parent (in $T$) of $N$ (but excluding the lower-level source trie (if any) that hangs from $N$) under the constraint that the root of the $2DHSST$ is an SST supernode whose utilization is $k + 1$. For example, when $k = 0$, the root of the constrained $2DHSST$ has a utilization 1 and contains only the parent of $N$; the remaining supernodes of the $2DHSST$ represent $2DBT(N)$.

Thus $f(N, h, k) = opt_1(N, h, k)$ when $k > 0$ and $1 + opt_1(N, h - 1, 0)$ when $k = 0$.

When $N$ has two children $a$ and $b$,

$$opt_1(N, h, k) = \min_{m(N) \leq i < h} \{ \min_{0 \leq j < k} \{ f(a, h - i, j) + f(b, h - i, k - j - 1) - 1 \} + s(N, i) \} \tag{14}$$

For $h \leq 0$

$$opt_1(N, h, \ast) = \infty \tag{15}$$

When we have $n$ filters and the length of the longest prefix is $W$, the number of nodes in the dest trie of $T$ is $O(nW)$ and the number of source tries in $T$ is $O(n)$. The time to compute all $s(N, h)$ values using the algorithm described in Section 4 to compute $opt$ is $O(n^2 WHK^2)$ time. Using Equation 10 and previously computed $g$ values, $O(H)$ time is needed to compute each $g(\ast, \ast, \ast)$ value. Using Equation 11, each $opt_1(\ast, \ast, \ast)$ value may be computed in $O(K)$ time. Using Equations 12-15, we can compute each $opt_1(\ast, \ast, \ast)$ value in $O(KH)$ time. Since there are $O(nWH)$ $opt_1(\ast, \ast)$, $O(nWHK)$ $opt_1(\ast, \ast, \ast)$, and $O(nWSH)$ $g(\ast, \ast, \ast)$ values to compute, the time to determine $opt_1(root(T), H)$ is $O(n^2 WHK^2 + nWHK + nWH^2K^2 + nWSH^2) O(n^2 WHK^2)$ (as, in typical applications, $n > H$).

## 6 2DHSSTs With Prefix Inheritance (2DHSSTP)

Let $T$ be the 2DBT of Figure 6. Consider the dest-trie supernode ab of Figure 8. This supernode represents the subtrie of $T$ that is comprised of the binary nodes $a$ and $b$. A search in this subtrie has three exit points—left child of $b$, right child of $b$ and right child of $a$. For the first two exit points, the source tries that hang off of $a$ and $b$ are searched whereas for the third exit point, only the source trie that hangs off of $a$ is searched. We say that the first
two exit points use the source tries that hang off of a and b while the third exit point uses only the source trie that hangs off of a. If the source trie that hangs off of b is augmented with the prefixes in the source trie that hangs off of a, then when the first two exit points are used, only the augmented source trie that hangs off of b need be searched.

In prefix inheritance, each non-empty source trie in a partition is augmented with the prefixes in all source tries that hang off of ancestors in the partition. When this augmentation results in duplicate prefixes, the least-cost prefix in each set of duplicates is retained. The resulting augmented source tries are called exit tries. In a 2DHSST with prefix inheritance (2DHSSTP), prefix inheritance is done in each supernode. Figure 9 gives the 2DHSSTP for the 2DHSST of Figure 8.

Notice that to search a 2DHSSTP we need search at most one exit trie for each dest-trie supernode encountered—the last exit trie encountered in the search of the partition represented by that dest-trie supernode. So, when searching for \((da, sa) = (000, 111)\), we search the exit tries that hang off of b and g for 111. The number of memory accesses is 2 (for the two supernodes ab and dg) + 2 (1 to access the supernode in each of the two source tries searched) + 2 (to access the \(N\ H\) arrays for the source trie supernodes) = 6. The same search using the 2DHSST of Figure 8 will search three source tries (those hanging off of a, b, and g) for a total cost of 8 memory accesses.

A node \(N\) in a dest-trie partition is a dominating node iff there is an exit trie on every path from \(N\) to an exit point of the partition. Notice that if \(N\) has two children, both of which are dominating nodes, then the exit trie (if any) in \(N\) is never searched. Hence, there is no need to store this exit trie.

We are interested in constructing a space-optimal 2DHSSTP that can be searched with at most \(H\) memory accesses. Although we have been unable to develop a good algorithm for this, we are able to develop a good
algorithm to construct a space-optimal constrained 2DHSSTP for any 2DBT $T$. Note that the 2DHSSTP for $T$ is comprised of supernodes for the dest-trie of $T$ plus supernodes for the exit tries.

Let $2DHSSTPC(h)$ be a 2DHSSTP that is constrained so that (a) it can be searched with at most $h$ memory accesses and (b) the HSST for each exit trie is a minimum height HSST for that exit trie. Our experimental studies suggest that the space required by an HSST is somewhat insensitive to the height constraint placed on the HSST. So, we expect that the space required by a space-optimal $2DHSSTPC(h)$ is close to that required by a space-optimal $2DHSSTP(h)$.

Let $N$ be a node in the dest-trie of the 2DBT $T$ and let $opt_2(N, h)$ be the size of a space-optimal $2DHSSTPC(h)$ for the subtree, $ST(N)$, of $T$ rooted at $N$. The supernode strides are $K$ and $S$. Notice that $opt_2(root(T), H)$ gives the size of a space-optimal $2DHSSTPC(H)$ for $T$. The development of a dynamic programming recurrence for $opt_2$ follows the pattern used for our earlier dynamic programming recurrences. Suppose that the root of the space-optimal $2DHSSTPC(N)$ is a TBM supernode. Then,

$$opt_2(N, h) = 1 + ss(N) + \sum_{R \in Ds(N)} opt_2(R, h - 1 - h(R))$$

where $ss(N)$ is the sum of the sizes of the minimum height HSSTs for the exit tries of the root TBM supernode and $h(R)$ is the MNMA for the last exit trie (if any) of the root that is on the path to $R$; if there is no exit trie on this path, then $h(R) = 0$.

The only other possibility for the root of the $2DHSSTPC(h)$ is that it is an SST node whose occupancy is $k$ for some $k$ in the range $[1, K]$. Let $2DHSSTPC(N, h, k, p)$ be a $2DHSSTPC(h)$ for $ST(N)$ under the constraints (a) the root of the $2DHSSTPC$ is an SST node whose utilization is $k$ and (b) for the root, prefix inheritance is not limited to the partition of $T$ represented by the root of the $2DHSSTPC$; rather prefix inheritance extends up to the $p$ nearest ancestors of $N$ in $T$. Let $opt_2(N, h, k, p)$ be the size of a space-optimal $2DHSSTPC(N, h, k, p)$. We see that:

$$opt_2(N, h) = \min_{0 < k \leq K} \{ opt_2(N, h, k, 0) \}$$

To facilitate the computation of $opt_2(N, h, k, p)$, we introduce three new functions: $s(N, p)$, $h(N, p)$ and $x(N, h, k, p)$. If $N$ has a non-empty source trie, then $s(N, p)$ is the size of a space-optimal minimum-height HSST for the union of the source tries that hang off of $N$ and its $p$ nearest ancestors in $T$ and $h(N, p)$ is the MNMA for this HSST. Otherwise, $s(N, p) = h(N, p) = 0$. The $s(N, p)$ values are computed prior to this postorder traversal using the algorithm of Section 4. The $h(N, p)$ values are computed easily during the computation of the $s(N, p)$ values.

$x(N, h, k, p)$ is the size of a space-optimal $2DHSSTPC(N, h, k, p)$ under the added constraint that the root of the $2DHSSTPC(N, h, k, p)$ is a dominating node. We obtain recurrences for $opt_2(N, h, k, p)$ and $x(N, h, k, p)$ by considering three cases for $N$. When $N$ has no child (i.e., $N$ is a leaf),

$$opt_2(N, h, k, p) = \begin{cases} \infty & \text{if } k < 1 \text{ or } h < h(N, p) \\ 1 + s(N, p) & \text{otherwise} \end{cases}$$
\[ x(N, h, k, p) = \begin{cases} 
\infty & \text{if } k < 1 \text{ or } h < h(N, p) \text{ or } N \text{ has an empty source trie} \\
\text{opt}2(N, h, k, p) & \text{otherwise}
\end{cases} \] (19)

When \( N \) has a single child \( a \),

\[ \text{opt}2(N, h, k, p) = \begin{cases} 
1 + \text{opt}2(a, h - 1 - h(N, p)) + s(N, p) & \text{if } k = 1 \\
\min\{\text{opt}2(a, h - 1 - h(N, p)) + \text{opt}2(h, k - 2h - 1 + p + 1) + s(N, p), \\
\text{min}_{0<j<k-1}\{\text{opt}2(a, h, j, p + 1) + \text{opt}2(b, h, k - j - 1, p + 1)\} - 1 + s(N, p), \\
\text{min}x(N, h, k, p) & \text{otherwise}
\end{cases} \] (20)

where

\[ \text{min}x(N, h, k, p) = \min_{0<j<k-1}\{x(a, h, j, p + 1) + x(b, h, k - j - 1, p + 1)\} - 1 \] (23)

\[ x(N, h, k, p) = \begin{cases} 
\text{min}x(N, h, k, p) & \text{if } N \text{ has an empty source trie} \\
\text{opt}2(N, h, k, p) & \text{otherwise}
\end{cases} \] (24)

Combining Equations 16 and 17, we get

\[ \text{opt}2(N, h) = \min\{1 + ss(N) + \sum_{Q \in D_S(N)} \text{opt}2(Q, h - 1 - h(Q)), \min_{0<k\leq K}\{\text{opt}2(N, h, k, 0)\}\} \] (25)

When we have \( n \) filters and the length of the longest prefix is \( W \), the number of nodes in the dest trie of \( T \) and hence the number of exit tries is \( O(nW) \). Using the algorithm of Section 4, all \( s(\ast, \ast) \) and \( h(\ast, \ast) \) values may be computed in \( O(n^2W^2HK^2) \) time. Following this computation, each \( ss(N) \) value may be computed in \( O(2^S) = O(K) \) time by traversing the first \( S \) levels of the subtree of \( T \) rooted at \( N \). Thus all \( ss(\ast) \) values may be determined in \( O(nWK) \) additional time. As can be seen from Equation 25, \( O(K) \) time is need to compute each \( \text{opt}2(\ast, \ast) \) value (assuming that the \( ss \) and \( \text{opt}2 \) terms in the right-hand-side of the equation are known). It takes \( O(K) \) time to compute each \( \text{opt}2(\ast, \ast, \ast, \ast) \) and \( x(\ast, \ast, \ast, \ast) \) value. As there are \( O(nWK) \) \( \text{opt}2(\ast, \ast) \) values and \( O(n^2WK) \) \( \text{opt}2(\ast, \ast, \ast, \ast) \) and \( x(\ast, \ast, \ast, \ast) \) values, the total time complexity is \( O(n^2W^2HK^2 + nWK + nWHK + nW^2HK^2) = O(n^2W^2HK^2) \).

7 Implementation Considerations

7.1 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 2.1, the size of a TBM node is \( 2S + 1 + 2h - 1 \)
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 [40]). 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. [40] 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. [7]. An additional bit (called prefixBit) is added to each supernode. This bit is a 1 for a supernode $N$ if 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 using controlled leaf pushing, which is quite similar to the standard leaf pushing proposed in [6]. In each supernode, if its underlying subtree root doesn’t contain a next hop, we store in this root the next hop of its nearest ancestor that contains a next hop. This way we neither need a prefixBit nor need keeping track of $Z$. We call this enhanced prefix-bit implementation. The number of memory accesses for a search is $H + 3$.

### 7.2 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 \textit{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, \( NA_1 \) requires 162 bits, and \( NA_2 \) 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 \((CA_1, NA_1)\) and \((CA_2, NA_2)\) into memory uses \( n_1 + n_2 + 6 \) \( B \)-bit words, where \( n_1 \) and \( n_2 \) are, respectively, the number of supernodes in \( CA_1 \) and \( CA_2 \). If we flip \((CA_2, NA_2)\) to get \((NA_2, CA_2)\) then the next-hop array \( NA_2 \) can use 36 of the 54 bits of a \( B \)-bit word not used by \( NA_1 \) and reduce the total word count by 1. This sharing of a \( B \)-bit word by \( NA_1 \) and \( NA_2 \) leaves 18 unused bits in the shared \( B \)-bit word and the child array \( CA_2 \) remains aligned to a word boundary. The child pointer for \((NA_2, CA_2)\) now points to the start of the array \( NA_2 \) and to compute the start of the array \( CA_2 \) from this child pointer, we must know the number of next-hop entries in \( NH_2 \). 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 \( b_1, \ldots, b_n \) 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 \( b_i \)s 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 \( b_i \)s 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.

### 7.3 End-Node Optimized HSSTs

A further reduction in the space requirements of an HSST may be achieved by employing end-node optimization [7]. We permit four formats for a leaf supernode. Figure 10 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.

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 Figure 10, $len(S1)$ 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 $N$). For all $ST(N)$s that are represented by a leaf supernode, we set $opt(N, h) = 1$ for $h \geq 0$. The dynamic programming recurrence of Section 4 is then used to determine $opt(root(T), H)$. The fourth format (SuffixB) is similar to the third leaf supernode format, while avoids an additional access to extract the next hop. When controlled leaf pushing is applied to the fourth format leaf supernode, the worst-case number of memory accesses required for a lookup may be reduced. Note without controlled leaf pushing, if no matching prefix is found in a fourth format leaf supernode, we still need an additional access to extract the next hop associated with the longest matching prefix along the search path.

For all $ST(N)$s that 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 prefix-bit implementation as well to reduce total memory requirement.

### 7.4 2DHSSTs and 2DHSSTPCs

We use the enhanced base implementation of an HSST for both the dest and source tries of a 2DHSST and a 2DHSSTPC. End node optimization is done on each source trie of a 2DHSST and a 2DHSSTPC. For the dest trie, however, we do the following:

1. Cut off the leaves of the dest binary trie prior to applying the equations of Sections 5 and 6 to construct space-optimal 2DHSSTs and 2DHSSTPCs. Following the construction, identify the parent dest-trie supernode for each leaf that was cut off.

![Figure 10: Leaf supernode formats](image-url)
2. In the case of 2DHSSTPCs, each source trie that hangs off of a leaf of the dest binary trie, inherits the prefixes stored along the path, in the parent dest-trie supernode, to this leaf.

3. Each cut-off leaf is replaced by the HSST for its source trie (this source trie includes the inherited prefixes of (2) in case of a 2DHSSTPC). The root of this HSST is placed as the appropriate child of the parent dest-trie supernode. (This requires us to use an additional bit to distinguish between dest-trie supernodes and source HSST roots.)

By handling the leaves of the binary dest-trie as above, we eliminate the need to search the source tries that are on the path, in the dest-trie parent, to a leaf child.

Finally, for 2DHSSTPCs, we may reduce the time and space required to construct space-optimal structures by using an alternative definition of the $p$ used in Section 6. In this new definition, prefix inheritance extends up to the $p$ nearest ancestors of $N$ in $T$ that have a non-empty source trie. Since, on typical data sets, a dest-trie node has a small (say 3 or 4) number of ancestors that have non-empty source tries while the number of ancestors may be as large as 32 in IPv4 and 128 in IPv6, the new definition of $p$ allows us to work with much smaller $p$s. This reduces the memory required by the arrays for $x(*,*,*,*)$ and $opt2(*,*,*,*)$ and also reduces the computation time. Note that the equations of Section 6 have to be modified to account for this change in definition. Note also that while the space required for $minx(*,*,*,*)$ also is reduced, we may solve the recurrences of Section 6 without actually using such an array.

8 Experimental Results

C++ codes for our algorithms for space-optimal 1- and 2-dimensional supernode tries were compiled using the GCC 3.3.5 compiler with optimization level 03 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 1- and 2- higher-dimensional packet classification [40, 41, 36, 21, 22]. 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 classifier 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. In the case of two-dimensional tables, we need to store the priority and action associated with a prefix. We allocate 18 bits for this purpose.
8.1 One-Dimensional Routing Tables

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. [40] and the variant 3 algorithm (which we refer to as V3MT) of [41] to construct multi-way trees. Extensive experiments reported in [41] 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. [40] establishes the superiority of BFP over TBM [7]. However, [40] did not compare BFP to V3MT.

IPv4 Router Tables

For test data, we used both IPv4 and IPv6 router tables. 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 [30, 32, 13]. The number of prefixes in these router tables is 17486, 29608, 103555, 109600, 173501 and 215487, respectively.

<table>
<thead>
<tr>
<th>Database</th>
<th>EP</th>
<th>EPO</th>
<th>EB</th>
<th>BFP</th>
<th>V3MT</th>
<th>EP</th>
<th>EPO</th>
<th>EB</th>
<th>EBO</th>
<th>BFP</th>
<th>V3MT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aads</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>MaeWest</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>RRC01</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>RRC04</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>AS4637</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>AS1221</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 11: Number of memory accesses required for a lookup in IPv4 tables

Figure 11 shows 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 [40, 41], the numbers reported by us include an additional access needed to obtain the next hop for the longest matching prefix. Figure 12 plots this data. 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 indicate that EBO may require one memory access less than EPO when \( B=72 \)) and on 2 when \( B = 144 \). Figure 13(a) normalizes the access count data by the counts for EBO and presents the min, max, and standard deviation of the normalized count for the 6 data sets. 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.

\(^2\)The BFP algorithm of [40] 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 [40].

\(^3\)We are grateful to the authors of [41] for providing their code.
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.

Figure 14 shows the total memory required by the lookup structure constructed by each of our 6 algorithms. Figure 15 plots this data and Figure 13(b) presents statistics normalized by the data for EBO. As can be seen, EPO and EBO results in the least total memory requirement. The memory required by the EBO structures is very close to that required by the EPO structures (the EBO structures, on average, required 2\% less memory than did the EPO structures). 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 \( B \) goes from \( B = 72 \) to \( B = 144 \), on 4 of the 6 data sets the memories required by EPO decreased, while
increased on the remaining 2. The total memory require when $B = 144$ normalized by that required when $B = 72$ is between 95% and 126%, the average was 105%, and the standard deviation was 15%. For EBO, the memory also decreased on 4 of the 6 data sets and increased on the remaining 2. The corresponding normalized memory requirement were 96%, 113%, 100%, and 7%.

On our IPv4 data sets, EBO is the clear winner as far as number of memory accesses go; EPO is slightly superior to EBO on the memory measure on 9 of our 12 test cases while interior on the remaining 3. All things considered, EBO is to be preferred. 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%.

When $B = 72$, the average number of bits of storage needed per prefix is 48 for HSSTs (using the BFP algorithm of [40]), 42 for V3MT [41] and 27 for HSSTs (EBO). The corresponding numbers for the case when $B = 144$ are 41, 35, and 27.
Comparison With Other Succinct Representations

Degermark et al. [6] have proposed a succinct router table structure called Lulea. This is a 3-level multibit trie. A lookup in Lulea requires 12 memory accesses. So, as far as lookup time goes, Lulea is inferior to all 6 of the structures we have considered here. Since we do not have the code for Lulea, we are able to do only an approximate memory comparison. [6] reports memory requirements for 6 databases, the largest of which has 38,141 prefixes and uses 34 bits of memory per prefix. Since the memory required per prefix decreases with database size, we compare with MaeWest, which has 29,608 prefixes. On MaeWest, with \( B = 72 \), EPO, EBO, BFP and VM3T, respectively, require 32, 31, 55, and 49 bits per prefix. The corresponding numbers with \( B = 144 \) are 30, 35, 46, and 40. We note that this is a very approximate comparison for the following reasons (1) the databases are different and (2) the number of bits allocated to pointers and next hops is different in Lulea and the remaining structures. For example, the Lulea scheme requires the size of a pointer to be the same as that of a next hop and so allocates 14 bits to each. We use 22 bits for a pointer and 12 for a next hop. Reducing the next hop size to 12 bits in Lulea doesn’t reduce the total memory required unless we also reduce pointer size to 12 bits. Assuming these inequities balance out, the data suggest that EPO and EBO are superior to Lulea on both the lookup complexity and memory complexity!

Lunteren [26, 27] has proposed a succinct representation of a multibit trie using perfect hash functions - balanced routing table search (BARTs). Figure 16 gives the memory requirement of BARTs 12-4-4-4-8, one of his two most memory efficient schemes (the other scheme is BARTs 8-4-4-4-4-8, which requires slightly less memory but two more accesses for a search). The number of memory accesses needed for a lookup is 9 in BARTs 12-4-4-4-8. By comparison, the lookup complexity for EBO with \( B = 72 \) is 5 or 6 accesses/lookup, and the total memory required is between 38% and 43% the memory of BARTs 12-4-4-4-8. We note that the implementation assumptions used by Lunteren [26] and us are slightly different. [26] allocates 18 bits for each pointer and next hop whereas we allocate 22 bits for a pointer and 12 for a next hop. The scheme of [26] requires pointers and next hops to be of the same size. In reality, the number of different next hops is small and 12 bits are adequate. On the other hand, for large databases, 18 bits may not be adequate for a pointer. Despite these minor differences, our experiments show that EBO is superior to the scheme of [26] on both lookup complexity and total memory required.

<table>
<thead>
<tr>
<th>Database</th>
<th>Aads</th>
<th>MaeWest</th>
<th>RRC01</th>
<th>RRC04</th>
<th>AS4637</th>
<th>AS1221</th>
</tr>
</thead>
<tbody>
<tr>
<td>BARTs 12-4-4-4-8 (B=32)</td>
<td>163</td>
<td>262</td>
<td>793</td>
<td>844</td>
<td>1270</td>
<td>1685</td>
</tr>
<tr>
<td>BARTS 12-6-6-8 (B=288)</td>
<td>137</td>
<td>214</td>
<td>692</td>
<td>733</td>
<td>1149</td>
<td>1380</td>
</tr>
</tbody>
</table>

Figure 16: Memorys (KBytes) of BART search

Lunteren [27] describes perfect-hash-function strategies for very wide memories - balanced routing table search (BARTS), \( B \geq 288 \). Figure 16 shows the memory requirement of his most memory efficient scheme BARTS 12-6-6-8 with \( B = 288 \). The number of memory accesses needed for a lookup is 4. EBO with \( B = 144 \) achieves a lookup complexity of 3 or 4 accesses/lookup while requiring from 44% to 60% the memory of BARTS 12-6-6-8.
IPv6 Router Tables

For our IPv6 experiments, we used the 833-prefix AS1221-Telstra router table that we obtained from [30] as well as 6 synthetic IPv6 tables. Prefixes longer than 64 were removed from the AS1221-Telstra table as current IPv6 address allocation schemes use at most 64 bits [1]. For the synthetic tables, we used the strategy proposed in [44] to generate IPv6 tables from IPv4 tables. In this strategy, to each IPv4 prefix we prepend a 16-bit string comprised of 001 followed by 13 random bits. If this prepending doesn’t at least double the prefix length, we append a sufficient number of random bits so that the length of the prefix is doubled. Following this prepending and possible appending, we drop the last bit from one-fourth of the prefixes so as to maintain the 3:1 ratio of even length prefixes to odd length observed in real router tables. Each synthetic table is given the same name as the IPv4 table from which it was synthesized. The AS1221-Telstra IPv6 table is named AS1221* to distinguish it from the IPv6 table synthesized from the IPv4 AS1221 table.

<table>
<thead>
<tr>
<th>Database</th>
<th>EP</th>
<th>EPO</th>
<th>EB</th>
<th>EBO</th>
<th>BFP</th>
<th>V3MT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1221*</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Aads</td>
<td>7</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>MaeWest</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>RRC01</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>RRC04</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>AS4637</td>
<td>8</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>AS1221</td>
<td>9</td>
<td>6</td>
<td>9</td>
<td>7</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 17: Number of memory accesses required for a lookup in IPv6 tables

![Figure 17](image1.png)

Figure 18: Number of memory accesses required by a lookup in IPv6 tables

![Figure 18](image2.png)

Figures 17 and 20 give the number of memory accesses and memory required by the search structures for our 7
### Table 1: Statistics for IPv6 data normalized by EBO data

<table>
<thead>
<tr>
<th>Database</th>
<th>EP</th>
<th>EPO</th>
<th>EB</th>
<th>EBO</th>
<th>BFP</th>
<th>V3MT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS1221*</td>
<td>7255</td>
<td>4372</td>
<td>6574</td>
<td>4755</td>
<td>8634</td>
<td>10996</td>
</tr>
<tr>
<td>Aads</td>
<td>473</td>
<td>170</td>
<td>391</td>
<td>184</td>
<td>501</td>
<td>558</td>
</tr>
<tr>
<td>MaeWest</td>
<td>769</td>
<td>296</td>
<td>641</td>
<td>311</td>
<td>836</td>
<td>946</td>
</tr>
<tr>
<td>RRC01</td>
<td>2581</td>
<td>974</td>
<td>2102</td>
<td>1046</td>
<td>2785</td>
<td>3267</td>
</tr>
<tr>
<td>RRC04</td>
<td>2744</td>
<td>1030</td>
<td>2234</td>
<td>1108</td>
<td>2969</td>
<td>3459</td>
</tr>
<tr>
<td>AS4637</td>
<td>4249</td>
<td>1617</td>
<td>3449</td>
<td>1752</td>
<td>4609</td>
<td>5445</td>
</tr>
<tr>
<td>AS1221</td>
<td>5656</td>
<td>2182</td>
<td>4665</td>
<td>2252</td>
<td>6240</td>
<td>6982</td>
</tr>
</tbody>
</table>

### Figure 19: Statistics for IPv6 data normalized by EBO data

IPv6 data sets. Figures 18 and 21 plot these data and Figure 19 gives statistics normalized by the data for EBO. EPO and EBO are the best with respect to number of memory accesses. When $B = 72$, EPO was superior to EBO by 1 memory accesses on 2 of the 7 data sets and tied on the remaining 5. However, when $B = 144$, EBO was superior to EPO by 1 memory accesses on 3 of the 7 data sets and tied on the remaining 4. As with the IPv4 data,
the memory utilization of the EBO structures is almost as good as of the EPO structures (an average difference of 1%). Worst-case lookups in the constructed BFP structures require 1.14 to 2.00 times as many memory accesses as required in the EBO structures and the BFP structures require 1.82 to 3.17 times the memory required by the EBO structures.

As was the case for our IPv4 experiments, increasing $B$ from 72 to 144, results in a reduction in the number of memory accesses required for a lookup. For EPO the maximum, minimum, and average reduction in the number of memory accesses were 33%, 17%, and 25%; the standard deviation was 8%. The corresponding percentages for EBO were 57%, 20%, 34%, and 13%. The memory of EPO decreased on 5 of the 7 data sets while increased on the remaining 2. The total memory require when $B = 144$ normalized by that required when $B = 72$ is between 86% and 110%, the average was 101%, and the standard deviation was 10%. For EBO, the memory requirement decreased on all of the 7 data sets. The maximum, minimum, and average memory reduction were 16%, 4%, and 12%; the standard deviation was 4%.

### 8.2 Further Optimizations

Song et al. [40] have proposed two techniques–child promotion and nearest-ancestor collapse, that may be used to reduce the number of nodes and number of prefixes in the one-bit binary tree. These techniques reduce the size of the one-bit binary trie as well as that of its compact representation. In child promotion, we promote the prefix stored in a binary node, if its sibling also contains a valid prefix, to the parent node. After the promotion, the node is deleted provided it is a leaf. In the nearest ancestor collapse technique, which draws from [18], we eliminate the prefix stored in a node if its nearest ancestor contains a prefix with the same next hop; leaves are deleted if they become empty. We note that nearest-ancestor collapse is very similar to the port merge technique proposed by Sun et al. [41]. A port merge is used to reduce the number of endpoints by merging two consecutive destination-address intervals that have the same next hop.

In this section, we study the effect of child promotion and nearest-ancestor collapse on the succinct representations generated by EBO, BFP, and V3MT. For V3MT, we do a port merge on the intervals constructed from the optimized binary trie. For this experimental study, we are able to use only 3 of our IPv4 data sets–Aads, Maewest, and AS1221 as these are the only data sets for which we have next-hop data.

<table>
<thead>
<tr>
<th>Database</th>
<th>$B = 72$ bits</th>
<th></th>
<th></th>
<th></th>
<th>$B = 144$ bits</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mem</td>
<td>MAs</td>
<td>Mem</td>
<td>MAs</td>
<td>Mem</td>
<td>MAs</td>
<td>Mem</td>
<td>MAs</td>
</tr>
<tr>
<td>Aads</td>
<td>51</td>
<td>5</td>
<td>94</td>
<td>7</td>
<td>72</td>
<td>7</td>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>MaeWest</td>
<td>85</td>
<td>5</td>
<td>151</td>
<td>7</td>
<td>116</td>
<td>8</td>
<td>89</td>
<td>3</td>
</tr>
<tr>
<td>AS1221</td>
<td>425</td>
<td>6</td>
<td>716</td>
<td>8</td>
<td>507</td>
<td>9</td>
<td>411</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1: Total memory (KBytes) and number of memory accesses (MAs) required by IPv4 tables after optimizations

Table 1 gives the total memory requirement and memory accesses needed for a lookup. EBO remains the best succinct representation method on both the number of memory accesses measure and the total memory measure.
On 2 of our 18 tests, (BFP on AS1221 with $B=144$ and V3MT on Aads with $B=72$), the number of memory accesses required for a lookup is reduced by 1. For the remaining 16 tests, there is no change in the number of accesses required for a lookup.

The application of the child promotion and nearest-ancestor collapse optimizations reduces the total memory required by the succinct representations of the binary trie. For EBO, the reduction varies from 24% to 35% with the mean reduction being 29%; the standard deviation is 5%. For BFP, these Percentages were 23%, 37%, 28% and 6%. These percentages for V3MT were 34%, 49%, 40%, and 7%. Our experiments indicate that most of the reduction in memory requirement is due to the nearest-ancestor collapse optimization. Child promotion contributed around 1% of the memory reduction.

The memory required by the BFP structures normalized by that required by the EBO structures was between 1.41 and 1.82, with the mean and standard deviation being 1.62 and 0.17. The corresponding ratios for V3MT were 1.02, 1.40, 1.19 and 0.16.

8.3 Multi-Dimensional IPv4 Tables

We evaluated the performance of our proposed data structures using both 2-dimensional and 5-dimensional data sets. We used twelve 5-dimensional data sets that were created by the filter generator of [42]. Each of these data sets actually has 10 different databases of rules. So, in all, we have 120 databases of 5-dimensional rules. The data sets, which are named ACL1 through ACL5 (Access Control List), FW1 through FW5 (Firewall), IPC1 and IPC2 (IP Chain) have, respectively, 20K, 19K, 19K, 12K, 19K, 19K, 18K, 17K, 19K, and 20K rules, on average, in each database. Our 2-dimensional data sets, which were derived from these 5-dimensional data sets, have, respectively, 20K, 19K, 10K, 13K, 5K, 19K, 19K, 18K, 17K, 17K, 16K and 20K rules on average in each database. The 2-dimensional rules were obtained from our 5-dimensional rules by stripping off the source and destination port fields as well as the protocol field; the dest and source prefix field were retained. Following this stripping process, duplicates were deleted (i.e., two rules are considered duplicate if they have the same dest prefix and the same source prefix).

8.3.1 Two-Dimensional IPv4 Tables

First, we compare the space-optimal minimum-access 2DHSST and 2DHSSTPC structures. Figure 22 and Figure 23 show the results from our experiment. For 5 of our 12 data sets–ACL2-5, and IPC1–2DHSSTPCs reduce the number of accesses at the expense of increased memory requirement. For the remaining data sets, 2DHSSTPCs and 2DHSSTs require almost the same number of accesses and the same amount of memory.

Across all our data sets, 2DHSSTPCs required between 0% and 29% more memory than required by 2DHSSTs (the mean increase in memory required was 6% and the standard deviation was 9%). As noted earlier, although 2DHSSTPCs required more memory, they required a smaller number of memory accesses for a lookup. The reduction in number of memory accesses afforded by 2DHSSTPCs was between 0% and 41% (the mean reduction was 11% and the standard deviation was 13%).
When $B$ is increased from 72 to 144, for both 2DHSSTs and 2DHSSTPCs, the number of memory accesses required is reduced, but the total memory required is generally increased. For 2DHSSTs, the total memory required when $B = 144$ normalized by that required when $B = 72$ is between 0.98 and 1.50 (the mean and the standard deviation are 1.21 and 0.19); the number of memory accesses reduces by between 28% and 41% (the mean reduction is 30% and the standard deviation is 9%). For 2DHSSTPCs, the normalized memory requirement is between 1.04 and 1.49 (the mean and standard deviation are 1.23 and 0.16); the reduction in number of memory accesses ranges from 18% to 56% (the mean reduction and the standard deviation are 31% and 11%).

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Mem (B=72)</th>
<th>MAs (B=72)</th>
<th>Mem (B=144)</th>
<th>MAs (B=144)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL1</td>
<td>329</td>
<td>9</td>
<td>479</td>
<td>6</td>
</tr>
<tr>
<td>ACL2</td>
<td>256</td>
<td>13</td>
<td>379</td>
<td>9</td>
</tr>
<tr>
<td>ACL3</td>
<td>102</td>
<td>26</td>
<td>102</td>
<td>16</td>
</tr>
<tr>
<td>ACL4</td>
<td>130</td>
<td>34</td>
<td>131</td>
<td>20</td>
</tr>
<tr>
<td>ACL5</td>
<td>40</td>
<td>17</td>
<td>39</td>
<td>10</td>
</tr>
<tr>
<td>FW1</td>
<td>236</td>
<td>11</td>
<td>274</td>
<td>8</td>
</tr>
<tr>
<td>FW2</td>
<td>261</td>
<td>11</td>
<td>392</td>
<td>8</td>
</tr>
<tr>
<td>FW3</td>
<td>210</td>
<td>11</td>
<td>260</td>
<td>8</td>
</tr>
<tr>
<td>FW4</td>
<td>203</td>
<td>11</td>
<td>237</td>
<td>8</td>
</tr>
<tr>
<td>FW5</td>
<td>216</td>
<td>11</td>
<td>274</td>
<td>9</td>
</tr>
<tr>
<td>IPC1</td>
<td>174</td>
<td>20</td>
<td>189</td>
<td>12</td>
</tr>
<tr>
<td>IPC2</td>
<td>265</td>
<td>10</td>
<td>300</td>
<td>8</td>
</tr>
</tbody>
</table>

Figure 22: Total memory (KBytes) and number of memory accesses (MAs) required by 2DHSSTs and 2DHSSTPCs

![Figure 22: Total memory (KBytes) and number of memory accesses (MAs) required by 2DHSSTs and 2DHSSTPCs](image)

Since our primary objective is to reduce the number of memory accesses, we use 2DHSSTPCs with $B = 144$ for
further benchmarking with 2DMSAs and 2DMTs [22]. The 2DMSAs and 2DMTs employed by us used the compression techniques packed array [38] and butler node [17]. These two techniques are very similar; both attempt to replace a subtrie with a small amount of actual data (prefixes and pointers) by a single node that contains these data. We note that 2DMTs and 2DMSAs are the best of the structures developed in [22], and using these two compression techniques, [24] has established the superiority of 2DMTs and 2DMSAs over other competing packet classification structures such as Grid-of-Tries [36], EGT-PCs [4], and HyperCuts [21]. For this further benchmarking, we constructed space-optimal 2DHSTPCs with the minimum possible number, $H$, of memory accesses for a worst-case search. This minimum $H$ was provided as input to the 2DMTsa (2DMTd) algorithm to construct a 2DMTsa (2DMTd) that could be searched with $H$ memory accesses in the worst case. Because of this strategy, the worst-case number of memory accesses for 2DHSTPCs and 2DMSAs (2DMTd) is the same.

![Figure 24: Total memory (KBytes) required by 2DHSTPCs, 2DMSAs, and 2DMTs](image)

Figure 24 plots the memory required by 2DHSTPCs, 2DMSAs, and 2DMTs. We see that, on the memory criterion, 2DHSTPCs outperform 2DMSAs by an order of magnitude, and outperform 2DMTs by an order of magnitude on 4 of our 12 data sets. The memory required by 2DMTs normalized by that required by 2DHSTPCs is between 1.14 and 624, the mean and standard deviation being 56 and 179. The normalized numbers for 2DMSAs were 9, 49, 17, 11. We also observed that when 2DMTs are given up to 60% more memory than required by space-optimal DHSSTPCs with the minimum possible $H$, we can construct 2DMTs that can be searched with 1 or 2 fewer accesses for our data sets FW1-5 and IPC2.

### 8.3.2 Five-Dimensional IPv4 Tables

For 5-dimensional tables, we extend 2DHSTPCs using the bucket scheme proposed in [4]. We start with a 2-dimensional trie for the destination and source prefixes. All rules that have the same dest-source prefix pair $(dp, sp)$ are placed in a bucket that is pointed at from the appropriate source trie node of the 2-dimensional trie. Since
\(dp\) and \(sp\) are defined by the path to this bucket, the dest and source prefix fields are not stored explicitly in a bucket. However, the source port range, dest port range, protocol type, priority and action are stored for each rule in the bucket. The 2DHSSTPC algorithms of this paper are used to obtain a supernode representation of the 2-dimensional trie and the \(NH\) lists of next-hop data are comprised of buckets. We modify \textit{SuffixB} nodes (an end-node optimization proposed in [25]) so that they contain source prefix suffixes, dest and source ports, protocols, priorities and actions rather than just source prefix suffixes, priorities and actions. During prefix inheritance in 2DHSSTPCs, a source trie may inherit prefixes, from its ancestor tries, that already are in that source trie. When this happens, the rules associated with these inherited prefixes need also to be stored in this source trie. To avoid this redundancy, we store a pointer in the bucket associated with a source-trie prefix, which points to the bucket associated with the same prefix in the nearest ancestor source trie. 2DHSSTPCs with buckets are called extended 2DHSSTPCs. Unlike 2DHSSTs, we do not modify the source tries of an extended 2DHSSTPC so that the last source prefix seen on a search path has highest priority (or least cost).

Florin et al. [4] state that when 2-dimensional tries with buckets are used, as above, for 5-dimensional tables, most buckets have no more than 5 rules and no bucket has more than 20 rules. While this observation was true of the data sets used in [4], some buckets had significantly more rules for our data sets. For example, in FW4, about 100 rules contain wildcards in both the dest and source prefix fields. These rules may be removed from the original data set and stored in a search structure that is optimized for the remaining 3 fields. We note that this strategy of storing a large cluster of rules with wildcards in the dest and source prefix fields in a separate structure was used earlier in the HyperCuts [21] scheme. The data reported in the following figures and tables are only for structures constructed for the rules that remain after rules with wildcards in both dest and source prefix fields are removed.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Mem</th>
<th>MAs</th>
<th>bits/rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL1</td>
<td>480</td>
<td>6</td>
<td>196</td>
</tr>
<tr>
<td>ACL2</td>
<td>392</td>
<td>12</td>
<td>169</td>
</tr>
<tr>
<td>ACL3</td>
<td>181</td>
<td>35</td>
<td>78</td>
</tr>
<tr>
<td>ACL4</td>
<td>252</td>
<td>32</td>
<td>108</td>
</tr>
<tr>
<td>ACL5</td>
<td>87</td>
<td>20</td>
<td>59</td>
</tr>
<tr>
<td>FW1</td>
<td>282</td>
<td>8</td>
<td>121</td>
</tr>
<tr>
<td>FW2</td>
<td>391</td>
<td>8</td>
<td>168</td>
</tr>
<tr>
<td>FW3</td>
<td>264</td>
<td>8</td>
<td>120</td>
</tr>
<tr>
<td>FW4</td>
<td>302</td>
<td>9</td>
<td>145</td>
</tr>
<tr>
<td>FW5</td>
<td>280</td>
<td>9</td>
<td>135</td>
</tr>
<tr>
<td>IPC1</td>
<td>244</td>
<td>23</td>
<td>105</td>
</tr>
<tr>
<td>IPC2</td>
<td>326</td>
<td>8</td>
<td>133</td>
</tr>
</tbody>
</table>

Figure 25: Total memory (KBytes), bits/rule, and number of memory accesses required by extended 2DHSSTPCs on 5-dimensional data sets.

Figure 25 gives the total memory and number of memory accesses required by extended 2DHSSTPCs on our twelve 5-dimensional data sets. Figure 26 compares 2DHSSTPCs (these, of course, store only the derived 2-dimensional rules) with extended 2DHSSTPCs that store 5-dimensional rules. The number of bits per rule required
by extended 2DHSSTPCs was between 59 and 196; the average was 128. Surprisingly, the addition of three fields increased the number of bits/rule by between 0.5 and 42 only; the average increase was only 13. In fact, for 8 of our 12 data sets (ACL1-2, FW1-5, and IPC2), extended 2DHSSTPCs and 2DHSSTPCs (both using \( B = 144 \)), required almost the same number of bits/rule. The very small increase in bits/rule is due to (a) port ranges in very many of our rules are wildcards that need only a bit each and (b) most of the data for the 3 additional fields in 5-dimensional rules can be packed into space in \( SuffixB \) nodes that is wasted in the case of 2-dimensional rules.

The number of memory accesses required to search our extended 2DHSSTPCs ranged from 6 to 35; the average was 15. For 6 of our 12 data sets (ACL1, FW1-3, FW4, and IPC2), there was no increase in the number of memory accesses required for a lookup in an extended 2DHSSTPC for a particular 5-dimensional data set versus a lookup in the 2DHSSTPC for the corresponding 2-dimensional data set.

HyperCuts [21], which is one of the previously best known algorithmic schemes for multidimensional packet classification, uses a decision tree and rules are stored in buckets of bounded size; each bucket is associated with a tree node. Unlike the bucket scheme used by extended 2DHSSTPCs in which the dest and source prefixes are not stored explicitly, the bucket scheme of HyperCuts requires the storage of these fields as well as those stored in extended 2DHSSTC buckets. So, the storage of an individual rule in HyperCuts requires more space than is required in extended 2DHSSTPCs. Additionally, in HyperCuts, a rule may be stored in several buckets whereas in extended 2DHSSTPCs, each rule is stored in exactly 1 bucket. The most efficient Hypercut scheme reported in [21] is HyperCuts-4. We use this scheme for comparison with extended 2DHSSTPCs.

Figure 27 shows the total memory and number of memory accesses required by HyperCuts [21], on our twelve 5-dimensional data sets. The number of bits per rule required by the HyperCuts structure was between 242 and 163,519; the average was 56,801. It is important to note that there is wide variation in the bits/rule required by

Figure 26: Total memory (KBytes) and number of memory accesses required by 2DHSSTPCs and extended 2DHSSTPCs

Figure 27: Total memory (KBytes) and number of memory accesses required by HyperCuts [21], on our twelve 5-dimensional data sets.
<table>
<thead>
<tr>
<th>Data Set</th>
<th>Mem</th>
<th>MAs</th>
<th>bits/rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL1</td>
<td>605</td>
<td>16</td>
<td>242</td>
</tr>
<tr>
<td>ACL2</td>
<td>10487</td>
<td>24</td>
<td>4415</td>
</tr>
<tr>
<td>ACL3</td>
<td>19591</td>
<td>43</td>
<td>8248</td>
</tr>
<tr>
<td>ACL4</td>
<td>17661</td>
<td>44</td>
<td>7436</td>
</tr>
<tr>
<td>ACL5</td>
<td>600</td>
<td>44</td>
<td>400</td>
</tr>
<tr>
<td>FW1</td>
<td>308121</td>
<td>26</td>
<td>129735</td>
</tr>
<tr>
<td>FW2</td>
<td>189751</td>
<td>24</td>
<td>79895</td>
</tr>
<tr>
<td>FW3</td>
<td>341813</td>
<td>23</td>
<td>151916</td>
</tr>
<tr>
<td>FW4</td>
<td>199096</td>
<td>24</td>
<td>93692</td>
</tr>
<tr>
<td>FW5</td>
<td>347478</td>
<td>20</td>
<td>163519</td>
</tr>
<tr>
<td>IPC1</td>
<td>38863</td>
<td>51</td>
<td>16363</td>
</tr>
<tr>
<td>IPC2</td>
<td>64394</td>
<td>24</td>
<td>25757</td>
</tr>
</tbody>
</table>

Figure 27: Total memory (KBytes), bits/rule, and number of memory accesses required by HyperCuts on 5-dimensional data sets.

Figure 28: Total memory (KBytes) and number of memory accesses required by HyperCuts and extended 2DHSSTPCs

Hypercutes; the bits/rule required by extended 2DHSSTPCs is far better predictable. In particular, [21] reports that the performance of HyperCuts is not good for firewall-like databases as these tend to have a high frequency of wildcards in the source and/or dest fields. In fact, [21] reports that a 10% presence of wildcards in either the source or dest prefix fields resulted in a steep increase in memory requirement! This observation is confirmed by our experiments. HyperCuts exhibited its best bits/rule performance on ACL1 and ACL5 (242 and 400, respectively), in which the frequency of wildcards in either the source or dest fields is less than 1%. It exhibited its worst performance on our 5 firewall data sets FW1-5 and on IPC2 (bits/rule ranged from 25,757 to 163,519). The wildcard frequency was between 60% and 90% in these data sets. The remaining data sets (ACL2-4 and IPC1) had a wildcard frequency between 10% and 15% and the bits/rule required by the Hypercuts structure varied from
The number of accesses required to search the Hypercuts structure for our data sets ranged from 16 to 51, with the average being 30.

Figure 28 compares extended 2DHSSTPCs and HyperCuts. The structure constructed by extended 2DHSSTPCs required between 0.1% and 79% the memory required by that constructed by HyperCuts; the average and standard deviation being 8% and 23%, respectively. The number of accesses for a lookup in the extended 2DHSSTPCs structure was between 31% and 81% that required by the HyperCuts structure; the average and standard deviation were 46% and 16%, respectively. For both schemes, the reported memory and accesses are only for the rules that remain after rules with wildcards in both dest and source prefix fields are removed.

Since, in extended 2DHSSTPCs, no rule is stored twice while the same rule may be stored in several Hypercuts buckets (depending on the complexity of the rule set), the memory requirement of 2DHSSTPCs is better predicted and far less on average and worst-case data.

9 Conclusion

We have developed a fast algorithm to construct minimum-height SSTs. Our algorithm reduces the complexity of this construction from \( O(m^2) \) \cite{40} 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 and good 2DHSSTs and 2DHSSTPCs. 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 \cite{40}. Additionally, our EBO structures require between 24% and 44% less memory. Compared to the structures produced by the V3MT algorithm of \cite{41}, our EBO structures require between 25% and 40% fewer memory accesses and between 12% and 38% less memory. The memory access and storage requirements of EBO also are superior to those of Lulea \cite{6} and the perfect-hash-function schemes of \cite{26, 27}. Similar improvements were provided by EPO and EBO on IPv6 data.

For two-dimensional IPv4 data sets, 2DHSSTPCs result in fewer memory accesses but more memory requirement than do 2DHSSTs. Since, memory accesses per lookup is the primary optimization criterion, we recommend 2DHSSTPCs over 2DHSSTs. Given the same budget for the number of memory accesses, 2DHSSTPCs require between 0.2% and 88% of the memory required by 2DMTs of \cite{22}, and between 2% and 11% of the memory required by 2DMTSAs of \cite{22}. For 5-dimensional classifiers, extended 2DHSSTPCs require significantly less memory and significantly less memory accesses, both on average and in the worst-case, than required by HyperCuts \cite{21}.

References


[30] [http://bgp.potaroo.net](http://bgp.potaroo.net)


[32] Ris, Routing information service raw data, [http://data.ris.ripe.net](http://data.ris.ripe.net)


