Shape Shifting Tries for Faster IP Route Lookup

Haoyu Song, Jonathan Turner, John Lockwood
Applied Research Laboratory
Washington University in St. Louis
Email: {hs1,jst,lockwood}@arl.wustl.edu

Abstract

Some of the fastest practical algorithms for IP route lookup are based on space-efficient encodings of multi-bit tries [1, 2]. Unfortunately, the time required by these algorithms grows in proportion to the address length, making them less attractive for IPv6. This paper describes and evaluates a new data structure called a shape-shifting trie, in which the data structure nodes correspond to arbitrarily shaped subtrees of the underlying binary trie for a given set of address prefixes. The ability to adapt the node shape to the trie reduces the number of nodes that must be accessed to perform a lookup, especially for tries with large sparse regions. We give a fast algorithm for optimally dividing a trie into nodes so as to minimize the maximum lookup depth. We show that seven data structure accesses are sufficient for route tables with more than 150,000 IPv6 prefixes. This makes it possible to achieve wire-speed processing for OC192 link using a single QDRII SRAM chip.

1 Introduction

The growth of Internet traffic and the growing complexity of packet processing are placing extreme demands on the design of high performance routers. More flexible and efficient methods are needed to perform high performance packet classification and route lookup.

Longest Prefix Matching (LPM) is now a well-understood problem, for which there is a variety of high performance algorithmic solutions [3, 4, 2, 5]. However, the expected deployment of IPv6, and the use of LPM as a component within more general packet classification mechanisms [6, 7, 8] create new challenges, justifying continuing efforts to improve the performance of LPM algorithms.

Some of the most successful methods for LPM are essentially high performance variants of the basic binary trie. The simplest variant of the binary trie is a multibit trie, in which binary nodes are replaced with d-ary nodes for values of $d > 2$. This can dramatically reduce the number of memory accesses required, at the cost of less efficient use of memory. The *tree bitmap* algorithm [2] can be viewed as a clever encoding of a multibit trie that dramatically reduces the memory penalty associated with a naive implementation. For each node in a multibit trie, tree bitmap uses a pair of bit vectors to represent the subset of the "potential children" that are actually present and the prefixes associated with the given node. Children of a node are stored in consecutive memory locations, allowing each node to use just a single child pointer. Similarly, the next hop information associated with a node is stored in a group of consecutive memory locations, allowing use of a single pointer to reference the next hop information. This representation allows every node in the multibit trie to be represented with a small, constant-size record. A tree bitmap implementation with an initial on-chip table of $8K$ entries (covering the first 13 bits of the IP address) and a stride of five needs just four off-chip memory accesses to traverse an IPv4 trie, with one or two additional accesses to retrieve the next hop information.

Unfortunately, the time needed for trie-based lookup mechanisms grows linearly in the address length, making them less attractive for IPv6. Reference [5] describes an algorithm whose complexity grows logarithmically in the prefix length, making it much more attractive for IPv6. However, the algorithm is relatively complex to implement and its use of pre-computed markers to guide the search makes it difficult to support incremental update. An alternative approach is to extend trie-based algorithms to make them more efficient for longer address fields. The key observation needed to enable this is that as address lengths grow, the structure of the underlying binary trie intrinsically becomes more and more sparse. This provides an opportunity to use alternate encodings that better match the structure of the binary trie. The *Shape-Shifting Trie* (SST), which we develop in this paper, is constructed from nodes that correspond to arbitrarily shaped subtrees of the underlying binary trie. This allows the SST to conform to the structure of the underlying binary trie, significantly reducing the number of SST nodes that must be traversed to perform a lookup.

The rest of the paper is organized as follows. Sections

2 and 3 discuss the SST coding scheme and the IP lookup algorithm, respectively. Section 4 describes the SST construction algorithm. An improved hybrid algorithm is introduced in Section 5 and the reference implementation is discussed in Section 6. The algorithm performance is evaluated for both IPv4 and IPv6 tables in Section 7. Incremental update of SST is discussed in Section 8. In Section 9, we discuss two additional optimizations that remove redundancies from the underlying binary trie, providing additional performance improvement. Section 10 summarizes the related work and we conclude the paper in Section 11.

2 SST Representation

The nodes of an SST correspond to subtrees of the underlying binary trie, with up to K nodes, where K is a parameter of the data structure. Since these subtrees can have an arbitrary shape, each SST node includes a *shape bitmap* (SBM) that represents the subtree's shape. The encoding we use was described by Jacobson [9]. To encode a tree, we first augment the tree with additional *dummy nodes*. Each original node with no children, gets two dummy children. Each original node with one child gets one dummy child. We then associate a bit with each node in the augmented tree. The value of this bit is '1' for each of the original nodes and '0' for each of the dummy nodes. The shape bitmap consists of this set of bits, listed in breadth-first order. We omit the bit corresponding to the root, since this bit is always '1'. The shape bitmap for a tree with K original nodes has $2K$ bits and any tree with up to K nodes can be represented by a shape bitmap with $2K$ bits. We can also view the shape bitmap as associating two bits with each original node. These bits indicate which of the node's potential children are present in the tree. In our illustrations, we typically adopt this viewpoint, to avoid showing dummy nodes explicitly.

In addition to the shape bitmap, an SST node includes an *internal bitmap* (IBM) with K bits. This identifies which of the binary trie nodes has an associated prefix. An SST node also includes an *external bitmap* (EBM) with $K + 1$ bits that identifies which of the potential "exit points" from the subtree corresponds to an actual node in the underlying binary trie. The bits of the internal and external bitmaps are listed in breadth-first order of the corresponding nodes.

Each SST node also includes two pointers. The *child pointer* points to the first SST node that is a child of the given SST node. The *next hop pointer* points to the next hop information for the first binary trie node in the SST node for which there is a prefix. The children of a given SST node are stored in sequential memory locations (allowing us to access any of the children using the pointer to the first one). Similarly, the next hop information for all the nodes is stored in sequential locations, allowing us to access the next hop information for any binary node, given a pointer to the next hop information for the first binary node for which there is a prefix.

Figure 1 shows a binary trie that has been divided into subtrees of size less than or equal to three, along with the corresponding shape-shifting trie.

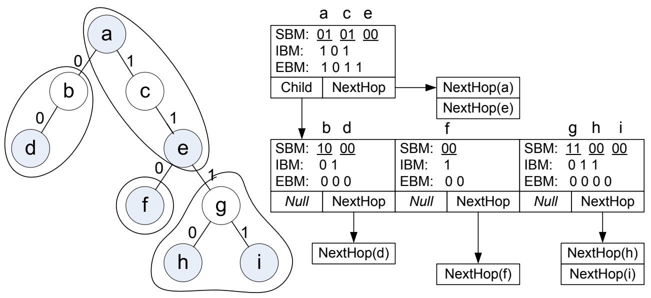

Figure 1. An SST with K=3 and the corresponding data structure. The darker binary trie nodes represent valid prefixes.

3 Lookup in an SST

The lookup process in an SST is similar to the lookup process for the tree bitmap algorithm [2]. The search proceeds recursively, starting from the root. At each step, we use bits from the address prefix to move through the subtree of the binary trie represented by the current SST node. We use the shape bitmap and external bitmap to allow us to determine if the search terminates at this node or continues to one of its children. If it does continue to a child, we find the bit in the external bitmap that corresponds to the child and count the number of '1's in the bitmap that precede this bit. We then use this number as an offset to the child node of interest, from the array of children starting at the location specified by the child pointer. An example illustrating this process is shown in Figure 2.

The basic step in the search algorithm requires decoding the shape bitmap. The key step is to find the bits in the shape bitmap that correspond to nodes in the path traversed by a search using bits from the IP address prefix. We start by defining n_i to be the number of nodes at distance i from the root of the augmented version of the subtree represented by the SST node (including dummy nodes). We let f_i denote the position of the bit in the shape bitmap that corresponds to the first node at distance i from the root. Note that $n_1 = 2$, $f_1 = 0$ (since we omit from the shape bitmap the bit corresponding to the root) and $f_i = f_{i-1} + n_{i-1}$. We define *ones*(i, j) to be the number of ones in the shape bitmap in the range of bits from i through j, and note that $n_i = 2 \times$ *ones*$(f_{i-1}, f_i - 1)$.

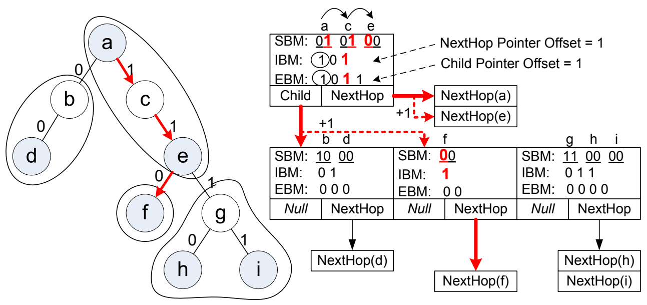

Figure 2. Assuming the IP address under lookup is "1100". The first SST node lookup returns the child pointer and the best matched prefix so far; the second SST node lookup returns the best matched prefix.

Next, we let a_i be the i-th bit of the IP address that is relevant to the node currently being decoded (so a_1 selects a child of the root of the subtree represented by the current node). We also let p_i be the index in the shape bitmap corresponding to the node on the path specified by the IP address that is at distance i from the root of the subtree. With these definitions, $p_1 = a_1$ and for $i > 1$, $p_i = f_i + 2 \times \mathit{ones}(f_{i-1}, p_{i-1} - 1) + a_i$.

Now, if i is the smallest integer for which the shape bitmap at position p_i is zero, then p_i corresponds to the point where the search on the IP address leaves the subtree represented by the current SST node. To determine if the search continues to another SST node, we need to consult the external bitmap. The position in the external bitmap that must be checked is the one with index equal to $\mathit{zeros}(0, p_i - 1)$ where $\mathit{zeros}(i, j)$ is defined to be the number of zeros in the shape bitmap in the range of bits from i through j. If x is the index of the proper bit in the external bitmap, and if bit x of the external bitmap is equal to '1', then the search continues at a child of the current SST node. To find the next SST node, we add an offset to the child pointer. This offset is equal to the number of ones in the external bitmap preceding bit x.

Consider the example shown in Figure 2. In the root SST node, we find $n_1 = n_2 = n_3 = 2$, $f_1 = 0$, $f_2 = 2$, $f_3 = 4$, $p_1 = 1$, $p_2 = 3$, $p_3 = 4$. Since bit p_3 of the shape bitmap is the first of the p_i bits that equals zero, we count the number of zeros in the shape bitmap preceding position 4. Since there are two zeros, we consult position 2 in the external map to determine if the search continue to another SST node. Since bit 2 of the external bitmap is 1, there is an extending path. Also, since there is a single 1 in the external bitmap before bit 2, we add 1 on to the child pointer to find the next SST node.

There are several ways to implement the lookup process for a single SST node. One conceptually simple approach is to use the equations derived above to define a combinational circuit that computes the values of p_i for $1 \leq i \leq K$. This is fast, but does require a relatively large amount of circuitry. A simpler alternative is to use a sequential circuit that for $i \geq 1$, computes values of n_i, f_i and p_i iteratively on successive clock ticks, terminating as soon as the shape bitmap at position p_i is equal to zero. This takes up to K clock ticks, plus another clock or two to decode the external bitmap and add the offset to the child pointer for the next memory access.

While the time needed to decode an SST node sequentially can be fairly long, note that the overall time to perform a lookup is essentially one clock tick per address bit, plus one memory access time per SST node searched. Since the lookup process does not change the SST, we can have multiple lookup engines operating in parallel on different packets, with their memory accesses interleaved. Thus, the time to do a lookup at a single node only affects the number of engines required, not the throughput. The throughput is a function only of the memory bandwidth and the number of memory accesses needed per lookup. See [10] for a description of how this technique is used with the tree bitmap algorithm of [2].

4 Constructing Optimal SSTs

A given binary trie can be represented by many different SSTs, depending on how the binary trie is partitioned. Since our primary concern is minimizing the search time, we focus on SSTs that have minimum height, where the height of a tree is defined as the length of a longest path from the root of the tree to a leaf.

However, we start by considering how to find an SST with a minimum number of nodes, ignoring the question of height. This can be done using a post-order traversal of the binary trie, pruning off subtrees to form SST nodes. Let $s(x)$ be the number of nodes in the subtree of the *current* binary trie with root x. When we visit node x in a post-order traversal, we perform the following step.

1. if $s(x) = K$, prune the subtree at x and assign all of its nodes to a new SST node.

2. otherwise, if $s(x) > K$ and x has children a and b with $s(a) \geq s(b)$, prune the subtree at a and assign its nodes to a new SST node.

We call this the *Post-Order Pruning* (POP) algorithm. Figure 3(a) shows an example of the partitioning produced by the POP algorithm for $K = 3$. Figure 3(b) also shows a minimum height partitioning. Notice that the minimum height partition has a height of one and yields five SST nodes, while the minimum size partition has a height of three and yields four SST nodes. The example makes it clear that a single SST cannot be optimal with respect to both criteria.

Theorem 1 *The SST constructed by the POP algorithm for a given binary trie has the minimum number of nodes.*

We sketch the proof of the optimality of the POP algorithm. We claim that the algorithm maintains the following invariant.

- *Invariant:* after every step, there is some SST with a minimal number of nodes that includes all the nodes formed so far.

This is clearly true when the algorithm starts and if it is true when the algorithm completes, then the constructed SST must be optimal. So, it suffices to show that the pruning rules maintain the invariant. Consider an application of the first pruning rule and let T be a minimum size SST that includes the nodes formed so far. If T does not form a node from the entire subtree at x, then at least one descendant of x must be in a different SST node than x is. This SST node cannot contain any nodes that are not descendants of x. Consequently, we can modify T so that it does form a single node from the subtree at x. The partition that T imposes on the rest of the binary trie remains unchanged. This SST cannot have any more nodes than T has.

Now, consider the second pruning rule. Again, let T be a minimum size SST that includes the nodes formed so far. Note that due to the post-order traversal, $K > s(a) \geq s(b)$. Because $s(x) > K$, T cannot form a single node from the subtree at x. Any subtree that is pruned from the subtree at x leaves behind at least $1 + s(b)$ nodes. Consequently, we can modify T so that it includes a node for the subtree at a, but is otherwise unchanged. This modified SST cannot have any more nodes than T. So far the optimality of the POP algorithm is proved.

We now turn our attention to constructing minimum height SSTs. This requires a somewhat more complicated method that we call the *Breadth-First Pruning* (BFP) algorithm. BFP operates in multiple steps, successively pruning off subtrees with at most K nodes. It starts by computing $s(x)$, the number of descendants of node x in the binary trie, for each binary trie node x. It then repeats the following step until there is nothing left of the binary trie.

- Scan the current pruned binary trie in breadth-first order. Whenever a binary trie node y with $s(y) \leq K$ is found, prune y and its descendants from the trie and assign them to a new SST node. For all ancestors x of y, subtract $s(y)$ from $s(x)$.

BFP can be implemented to run in $O(n^2)$ time, where n is the number of nodes in the underlying binary trie. We now show that it does produce minimum height SSTs.

Consider any minimum height SST for a given binary trie. We say that a binary trie node u "belongs" to an SST node x, if u is in the subtree corresponding to x. We assign each binary trie node u a label $h(u)$ equal to the height of the SST node it belongs to. To establish the optimality of the BFP algorithm we first prove a few properties concerning these labels.

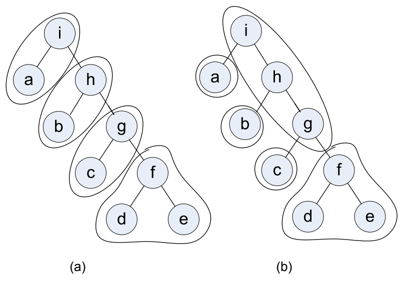

Figure 3. Minimum size and minimum height partitions of a binary tree.

Lemma 1 *For any node u, the number of descendants v of u (including u itself) with $h(v) = h(u)$ is at most K.*

Proof: Let S be the set of descendants v of u with $h(v) = h(u)$. Assume that S contains more than K nodes and note that, they cannot all belong to the same SST node. If U is the SST node that u belongs to, there must be some node v in S that belongs to a child V of U. But the height of U cannot equal the height of V, contradicting the assumption that S contains more than K nodes. □

We call each of the steps performed by the BFP algorithm a pass.

Lemma 2 *After i passes of the BFP algorithm, the binary trie contains no nodes u with $h(u) \leq i - 1$.*

Proof: Proof by induction. The basis $i = 0$ is trivially satisfied, since $h(u) \geq 0$ for all u.

For the inductive step, assume that at the beginning of pass i, the trie contains no nodes u with $h(u) \leq i - 2$. Suppose that at end of pass i, there is some node u with $h(u) = i - 1$. Since u was not removed from the trie, it must have already been considered in the breadth-first scan performed by BFP. Since it was not removed from the trie, it must have had more than K descendants at the time it was considered. But since all of its descendants v have $h(v) = i - 1$, this contradicts Lemma 1. □

Lemma 3 *Let x and y are two SST nodes formed by the BFP algorithm in the same pass, then neither is an ancestor of the other.*

Proof: The BFP algorithm scans the underlying binary trie in breadth-first order. In one pass, if a node is pruned, all of its ancestors have already been scanned and will not be touched again in the same pass. □

With these lemmas, we are now prepared to show that BFP produces minimum height SSTs.

Theorem 2 *The SST constructed by the BFP algorithm for a given binary trie has the minimum height. The height is one less than the number of passes performed by BFP.*

Proof: Let r be the root of the binary trie and let T be the SST constructed by BFP. By Lemma 2, the SST node containing r is formed by the end of pass $h(r)+1$. By Lemma 3, no path from the root of T to one of its descendants passes through more than one node formed in the same pass. Hence, the height of T is at most $h(r)$. Since $h(r)$ was defined relative to a minimum height SST, it follows that T has minimum height also. $\square$

5 A Hybrid Algorithm

The shape shifting trie method for longest prefix matching is a generalization of the tree bitmap algorithm (TBM) [2]. In both algorithms, the data structure node includes an internal bitmap, an external bitmap, a single child pointer and a single next hop pointer. However, SST also requires a shape bitmap that must be taken into account when comparing the two.

If we let $K = 2^S$ be the SST node size, then an SST node needs $4K+1$ bits for its three bitmaps. A TBM node can use these bits to implement a multibit trie node with a stride of $S+1$, corresponding to a subtree of the binary trie with $2K-1$ nodes. So, if the underlying binary trie is dense, the TBM data structure can be more space-efficient than the SST, but if the binary trie is sparse (fewer than half the ”potential” nodes are actually present), the SST is more space-efficient. Because such sparse subtrees are very common in the tries that represent large routing tables, SST is typically more space-efficient than TBM.

The more important advantage of SST is its potential to reduce the trie height. In the extreme case of a trie that consists of one long path with m modes, a TBM data structure has a height of approximately $m/(S+1)$ while a comparable SST has a height of $m/2^S$. For $S = 4$, this is more than a three-to-one improvement. In practice we don’t expect such dramatic gains, but we do find improvements as high as two-to-one for IPv6 in Section 7.

This discussion suggests that it may be worthwhile to use a hybrid approach in which TBM nodes are used to represent dense parts of the trie, while SST nodes are used to represent sparse parts. We use a bit in the node data structure to identify the format used for the current node. If the bit specifies a TBM node, we use $2K+1$ bits for the external bitmap, and $2K$ bits for the internal bitmap. If the bit specifies an SST node, we use $2K$ bits for the shape bitmap, $K+1$ bits for the external bitmap and K bits for the internal bitmap. The hardware required to decode the two types of nodes is simple enough that the cost of implementing both types of lookup is not a significant issue.

When building a hybrid trie, we must decide which node type to use. We modify the BFP algorithm to take this into account. During each breadth-first scan, when we encounter a node u, we first check to see if the height of the subtree with root u is small enough to allow it to fit in a TBM-type node. If it is, we prune the subtree and form a TBM-type node. Otherwise, we check to see if the number of nodes in the subtree is small enough to fit in a single SST node. If so, we prune the subtree and form an SST-type node. Note that whenever we encounter a node in a breadth-first scan, we already know that the height of its parent is too large for a TBM node and the size of its parent’s subtree is too large for an SST node. Also, note that the height of the hybrid data structure cannot be any larger than the height of an optimal SST. On the contrary, the hybrid data structure can potentially reduce the trie height further.

6 Reference Implementations

The performance evaluation to follow, is based on reference implementations of the TBM, SST and hybrid algorithms. We assume that in all three cases, the lookup data structure is stored in a 200MHz QDRII SRAM with a 36bit wide data interface. These devices have a minimum burst size of two words, which can be read and write in a single clock cycle. In our reference implementations, the nodes for each data structure are stored in three words. For the TBM data structure (and for TBM nodes in the hybrid data structure), this allows us to implement a stride of 5 (32 bits for the external bit map, 31 bits for the internal bitmap). For SST nodes, there is enough space for $K = 16$.

All three algorithms use a variation of the *prefix bit optimization* described in references [2, 10]. This optimization reduces the number of off-chip memory accesses substantially. It’s based on the observation that we don’t really need to look at the next hop pointer and the internal bitmap for most nodes visited during a search. We only need to examine these fields for the node corresponding to the longest prefix. The prefix bit optimization allows us to identify this node, without looking at the next hop pointer and internal bitmap fields of all but two nodes visited. The optimization is implemented using an extra bit in each data structure node. This bit is set to a ‘1’ if the portion of the underlying binary trie corresponding to the parent node has a prefix that is relevant to the child’s subtree. During the search, we remember the parent of the most recently visited node whose prefix bit was set. At the end of the search, we examine the next hop pointer and internal bitmap of this parent node. We also examine the next hop pointer and internal bitmap of the node where the search terminates. If all but the next hop pointer and internal bitmap are placed in the first two words of the three words used to store a data structure node, we only need to do one two-word access per data structure node visited, plus one or two more to retrieve the best matched next hop. Thus, if the data structure has a height of H, the worst-case number of memory accesses is $H + 3$.

These considerations lead to the node formats shown in Figure 4. In all cases, the third word contains the internal bitmap. For the SST node format it also contains the next hop pointer. Because the parameter K for an SST node does not have to be a power of two, one can increase the SST node size at the expense of reducing the number of bits in the child pointer. The child pointer size is set to 20. This allows us to have up to a million SST nodes. We allocate 20 bits to the next hop pointer, allowing for up to a million prefixes. Since the largest IPv4 prefix tables currently contain fewer than 200,000 prefixes, this seems more than adequate.

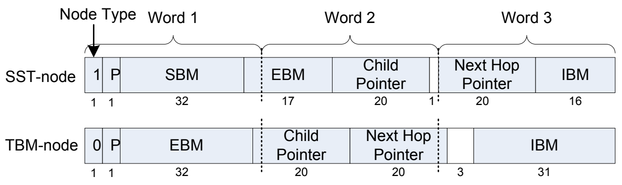

Figure 4. Data Structure Node Formats: K=16 for SST-type node and S=5 for TBM-type node

7 Performance Evaluation

Using our reference implementations, we performed simulations on real and synthetic IP route lookup tables to examine the performance of our algorithms in terms of tree height and tree size, which are determine the worst case lookup throughput and the memory consumption. Specifically, we compared three different algorithms: the *tree bitmap* algorithm, the original *BFP SST* algorithm and the *BFP hybrid* algorithm. We also provide the statistics of the underlying binary trie for reference. The parameter settings are as shown in Figure 4.

7.1 Performance on IPv4 Route Lookup

To start, we simulated the algorithms for IPv4 lookup tables. We expect the largest performance improvement on small tables since we can expect the prefix tree to be sparse and contain a lot of long and skinny paths. However, we are particularly interested in the algorithm performance on very large IP lookup tables. We used a recent snapshot of the $AS1221$ BGP table from [11] for analysis. This table contains about $184K$ prefixes and has the prefix length distribution shown in Figure 5.

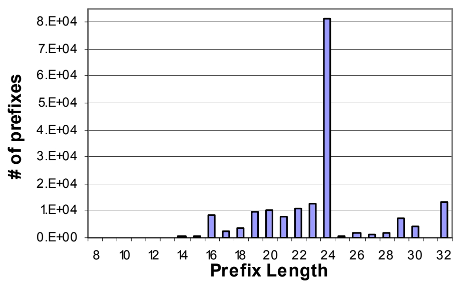

Figure 5. Prefix Length Distribution of IPv4 BGP Table

The prefixes lengths are distributed from 8 to 32. Almost half of the prefixes have length 24. Table 1 shows the test results.

Table 1. IPv4 BGP Table Results

	Trie Height	# of Nodes	Worst Case Throughput	Memory (Bytes)
Underlying Binary Trie	32	487,696	-	-
Tree Bitmap	6	64,245	22.2M pkts/s	845.0K
BFP SST	5	49,177	25.0M pkts/s	648.3K
BFP Hybrid	4	33,094	28.6M pkts/s	436.3K
SST Optimal Bound	5	37,760	25.0M pkts/s	497.8K

In summary, the BFP Hybrid algorithm improves the trie height by 33% and improves the trie size by 48% over the tree bitmap algorithm. The BFP SST algorithms reach the optimal trie height while the multibit trie is 1 layer taller. On the other hand, both of the BFP SST and BFP hybrid algorithms decrease the total number of nodes significantly compared with the tree bitmap algorithm. Surprisingly, the height and size of the trie generated by the BFP Hybrid algorithm are even lower than the optimal bound for the pure BFP SST case. In all cases, the tables are small enough to fit in a single SRAM chip (4 MB chips are currently available).

Assuming we fully utilize the memory bandwidth by deploying multiple lookup engines and interleaving the memory accesses, the BFP hybrid algorithm needs only 7 memory accesses in the worst-case, per route lookup. Since the QDRII SRAM can perform 200 million two-word accesses per second, it can sustain a throughput of 28.6 million packets per second. Assuming a worst case packet size of 40 bytes, the system can support 9.1 Gbps throughput, which is close to the OC192 link rate.

7.2 Performance on IPv6 Route Lookup

Evaluation is somewhat more difficult for IPv6, as there are no large real-world IPv6 routing tables available for analysis. We start with an available IPv6 BGP table from [11]. This table has fewer than 900 prefixes, with the prefix length distribution shown in Figure 6.

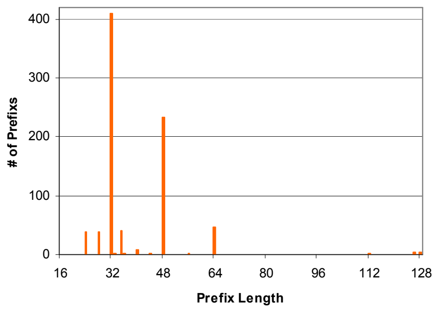

Figure 6. Prefix Length Distribution for IPv6 BGP Table

In this table, prefixes with length 32, 48 and 64 dominate and only a handful of prefixes have length of 112, 126 and 128 bits. Table 2 shows the test results. We note that the BFP SST and the BFP hybrid algorithms yield a trie height less than one third that required by the tree bitmap algorithm. This allows them to sustain a throughput that is almost three times higher.

Table 2. IPv6 BGP Table Results

	Trie Height	# of Nodes	Worst Case Throughput	Memory (Bytes)
Underlying Binary Trie	128	5,415	-	-
Tree Bitmap	25	1,013	7.14M pkts/s	13.4K
BFP SST	8	530	18.2M pkts/s	7.0K
BFP Hybrid	8	491	18.2M pkts/s	6.5K
SST Optimal Bound	8	413	18.2M pkts/s	5.4K

One can argue that this comparison is unrealistic, since current IPv6 address allocation schemes [12] use the lower half of the 128-bit IPv6 address for an interface ID. This makes it unnecessary to store more than the first 64 bits of the IP address prefix in the trie. To correct for this, we do a second comparison in which all prefixes with length longer than 64 have been removed. Table 3 summarizes the results. In this case, the BFP SST and BFP hybrid algorithms still provide more than a 2:1 reduction in the trie height and nearly a 2:1 reduction in the trie size, when compared to the tree bitmap algorithm.

To more fully evaluate our algorithms for the IPv6 case, we resort to synthetic IPv6 prefix sets, since there are no large real-world IPv6 tables available yet. We adopt the methodology developed in [13]. The authors observe that while it is difficult to predict the structure of future large scale IPv6 route lookup tables, it's possible to use the IPv6 address allocation schemes and the characteristics of current IPv4 tables to infer information that can be used to generate

Table 3. Trimmed IPv6 BGP Table Results

	Trie Height	# of Nodes	Worst Case Throughput	Memory (Bytes)
Underlying Binary Trie	64	5,015	-	-
Tree Bitmap	12	934	13.3M pkts/s	12.3K
BFP SST	5	498	25.0M pkts/s	6.6K
BFP Hybrid	5	455	25.0M pkts/s	6.0K
SST Optimal Bound	5	386	25.0M pkts/s	5.1K

realistic IPv6 tables. For evaluation, we generate an IPv6 table with about 200K prefixes using the method proposed in [13]. The prefix length distribution of this table is shown in Figure 7.

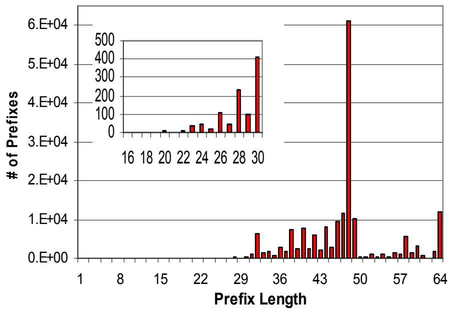

Figure 7. Prefix Length Distribution for Synthetic IPv6 Table

All the prefixes in this table are for global unicast addresses, which start with the first three bits of "001". The prefix length also retains some statistical characteristics of the IPv4 BGP table used in our earlier experiments, but scaled to IPv6. For example, the ratio of the number of even length prefixes to the number of odd length prefixes is 3 : 1. A large portion of the prefixes have length of 32, 48 and 64. This characteristic is also consistent with the IPv6 address allocation schemes and seems likely to hold true in the future IPv6 route lookup tables. Each address prefix is generated by starting with the three bit prefix 001, appending a 13 bit random number, then appending an IPv4 prefix, and finally appending some additional random bits whose length is selected to produce the desired prefix length distribution. The IPv4 prefixes were selected from the BGP table used in our earlier experiment.

The simulation results on this synthetic route lookup table are summarized in Table 4. The trie height for the BFP hybrid algorithm is about half that for the tree bitmap algorithm and the memory required is about 40% of that required by the tree bitmap algorithm. The pure BFP SST algorithm is only slightly less efficient than the BFP hybrid algorithm.

With similar numbers of prefixes, the binary tries for

Table 4. Performance on the Synthetic IPv6 Table

	Trie Height	# of Nodes	Worst Case Throughput	Memory (Bytes)
Underlying Trie	64	4,565,260	-	-
Tree Bitmap	12	892,111	13.3M pkts/s	11.8M
BFP SST	7	345,222	20.0M pkts/s	4.55M
BFP Hybrid	6	344,742	22.2M pkts/s	4.54M
SST Optimal Bound	7	312,132	20.0M pkts/s	4.12M

IPv6 route lookup tables are sparser than those for IPv4. Comparing the simulation results for the IPv4 and IPv6 route lookup tables, we note that the BFP hybrid algorithm makes a bigger difference in the space efficiency for the IPv4 case, apparently due to the greater density in the underlying trie.

For this scale of route lookup tables, we can finish 22.2 million route lookups per second. Assuming the worst case IPv6 packet size to be 60 bytes, a single SRAM chip can sustain $10.7 Gbps$ link speed.

While the height of trie-based data structures with a fixed stride length grows in proportion to the underlying binary trie height, we find that the SST height increases only by two as we go from IPv4 to IPv6. The number of memory accesses needed is actually comparable to the number of hash table probes needed for the method described in scheme [5]. The method of [5] requires $\log_2 n$ hash probes, where n is the address length, which is 64 for IPv6.

7.3 Scaling Characteristics of SST

We performed some additional experiments to show how the performance of SST improves as more bits are available for the per node bit maps. We used the synthetic IPv6 BGP table used in our earlier experiment and varied the *total* number of bits available for the bitmaps from 16 to 128. The results are summarized in Figure 8. At most of the data points, the SST algorithm shows substantial advantages over the tree bitmap algorithm.

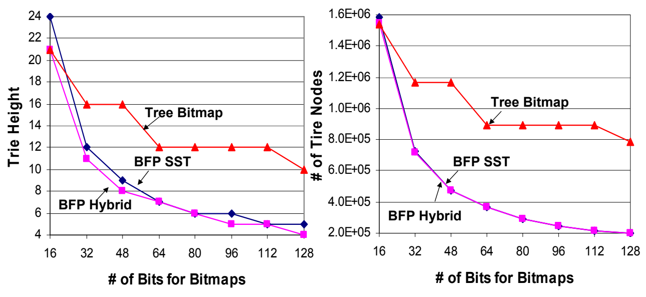

Figure 8. Effects of the Bit Assignment

8 Updating an SST

Because an SST is essentially an encoding of a binary trie, it is relatively easy to add and remove prefixes. Prefixes that convert a non-prefix node in the underlying binary trie to a prefix node are trivial to handle. It's also easy to add binary trie nodes to SSTs nodes that are not yet "full". In some cases, this may require restructuring an SST node, but so long as this restructuring does not change the set of child nodes of the SST node being restructured, it affects only the one SST node. Adding new SST nodes is also straightforward, as is removing SST nodes that are no longer needed.

However, incremental modifications to an SST can result in poor performance. In particular, one can construct a sequence of insertions and deletions that results in an SST with nodes that all have depth $\log_2 K$. This can lead to worst-case performance that is worse than that of the tree bitmap algorithm. One can avoid this by restructuring the SST occasionally, should the height exceed some target bound. Determining the frequency with which such restructuring should be done is left as a subject for future study.

9 Optimizations on Underlying Binary Trie

Since the size of the underlying binary trie directly correlates with the size of the SST, a smaller underlying trie is preferred, in order to reduce the memory usage. The real route lookup tables contain certain redundancies that can be exploited to compress the underlying binary trie. We briefly summarize two simple techniques called *child promotion* and *nearest ancestor collapse* that can be used to remove such redundancies.

Figure 9 illustrates the child promotion optimization. If two child nodes of a binary trie node both represent valid prefixes, we can use a one-bit shorter prefix to replace one of these two longer prefixes without changing the LPM lookup results. If this promoted child node is a leaf node, we can safely delete this child node after the promotion.

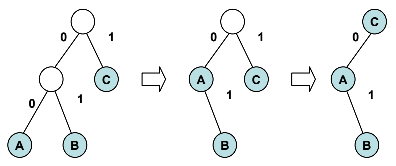

Figure 9. Child Promotion: the darker nodes represent the valid prefixes. We promote the prefix *A* in the first step and *C* in second step.

To perform this optimization, after building the binary prefix trie, we traverse the trie in the post order. If the two child nodes of a binary trie node are valid prefixes, we promote one of them to be the parent node. If one of the children is a leaf, we select it for promotion and delete the leaf. We find this optimization works quite well. It deletes 5.23% (5,426) of tree nodes and promotes 7,179 prefixes from the Mae-West route lookup table. For the much larger IPv4 BGP table used in our earlier experiments, it deletes 10.78% (52,585) of the tree nodes and promotes 74,804 prefixes.

Our second optimization is based on the observation that for route lookups, it does not matter which prefixes are matched, but rather what next hop is selected. The distinct next hops is typically much fewer than the prefixes, making it possible to eliminate certain prefixes. In general, if the next hop of a node in the binary trie is the same as the next hop of its nearest ancestor that corresponds to a prefix, we can eliminate the first prefix. Figure 10 illustrates the nearest ancestor collapse optimization.

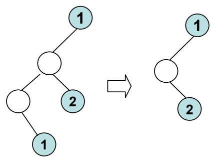

Figure 10. Nearest Ancestor Collapse: the darker nodes represent the valid prefixes and the number in the nodes indicates the next hop.

To perform this optimization, we traverse the binary trie in the post order. For each valid prefix, we examine its nearest ancestor that is also a valid prefix and compare their next hops. If they are same, we delete the next hop information in the longer prefix and invalidate this prefix. If this prefix node happens to be a leaf node, we recursively delete the nodes upwards until we meet its nearest ancestor or a tree branch.

This optimization decreases the number of binary trie nodes as well as the number of valid prefixes. Liu uses similar technique to compress the route lookup tables and shows that the number of valid prefixes can be reduced by up to 26.6% [14].

10 Related Work

IP route lookup is a well-studied problem. The algorithmic approaches organize the prefixes using some data structure and store them in memory. The lookup is conducted by a series of memory accesses. The multibit trie is representative of this type of approach. An alternative approach is to use a hardware-based search mechanism such as a Ternary Content-Addressable Memory (TCAM). TCAMs can deliver a search result per clock tick, making them very fast, but TCAM is also relatively expensive and consumes a great deal of power (both cost and memory consumption are more than times that of SRAM).

For trie-based IP lookup, some techniques have been developed to improve the lookup efficiency by exploiting the structural characteristics of the prefix tree. The original binary trie can have long sequence of nodes with single children. Path compression techniques [15] can be used to represent such long paths as a series of bits. The SST data structure can be viewed as a generalization of this approach.

The multibit trie is the more common technique to accelerate the IP route lookup speed. In this scheme, multiple bits are inspected simultaneously, so that the throughput is improved in proportion to the stride. One way to implement it is through the prefix expansion: arbitrary prefix lengths are transformed into an equivalent set with the prefix length allowed by the new structure. Specifically, if the stride is S, the prefix lengths that are not a multiple of S need to be expanded to make the lengths equal to the nearest multiple of S. The prefix expansion increases the memory consumption if the fixed stride is used.

The multibit trie algorithm is generalized to enable different stride at each trie level. Given the IP route lookup table and a desired number of memory accesses in the worst case, the selection of the stride size is implemented by the controlled prefix expansion [3]. A dynamic programming algorithm is used to compute the optimal sequence of strides that minimizes the storage requirements. The algorithm is further improved by providing alternative dynamic programming formulations for both fixed and variable-stride tries [16]. The disadvantages of the controlled prefix expansion are two folds: the update is slow and lead to suboptimality; the hardware implementation is difficult due to the variable trie node size. Moreover, the storage optimality is only under the worst case throughput constraints. The prefix expansion tends to increase the memory consumption anyway. For example, the memory usage of our data structure on the BGP table is roughly equal to the memory usage of the multibit trie with the controlled prefix expansion on the MaeEast table, when their worst-case tree heights are both five [3]. However, the BGP table is about five times larger than the MaeEast table. Clearly, our algorithm scales with the route table size much better.

The breakthrough to enable fast hardware implementation and eliminate the prefix expansion requirement is the tree bitmap algorithm [2]. The major idea also forms the foundation of our work. A coding scheme is used to effectively compress the node size and enable fast lookup. Another similar node coding scheme can be found in [17]: A depth-1 trie numbering scheme is actually a combination of the shape bitmap and the internal bitmap, while the external bitmap is implied by the trie scanning order. However, the major concern in that paper is to compress the trie representation. Though the data structure also supports multiple bit search in one memory access, it only uses naive trie partition and does not provide an optimal trie in terms of either height or size. Besides, the algorithm does not consider the case when the compression actually turns out to be more inefficient than the simple tree bitmap representation as addressed by our BFP hybrid algorithm.

In [18], the underlying binary trie is also partitioned to build the FSMs using the hardware logics for IP route lookups, where the partitioning is only aimed to reduce the overall number of FSMs. We also find the similarity of the SST construction problem with the technology mapping problem in the reconfigurable logic technology.

Many algorithms have been proposed to optimize the height and size of the partitions over the underlying binary tree, separately or simultaneously [19, 20]. Actually, the SST construction can be considered as a special and simplest case of this problem, hence these algorithms can be applied to our problem with corresponding modifications. Likewise, the BFP and POP algorithms, which are simple and optimal in terms of trie height or size, can also be used in technology mapping scenarios.

11 Conclusion

In this paper, we have presented a novel data structure, the *shape shifting trie* and an IP lookup algorithm that uses it. The algorithm outperforms the well-known and highly successful tree bitmap algorithm, and can be used in high performance routers to perform IP route lookup at even higher line speed. The algorithm also scales well to the fast growing route lookup table and is especially attractive for the IPv6 route lookups, since it is designed to exploit the intrinsic sparsity of the IPv6 prefix tree. We show how to structure SSTs to achieve minimum size and height (although not simultaneously). We also describe a hybrid algorithm that combines elements of the tree bitmap and SST approaches.

We show that using a single QDRII SRAM chip, an SST-based route lookup system can achieve wire-speed processing of IPv4 and IPv6 packets at OC192 link rates. Higher throughput can be obtained by increasing the memory bandwidth further. One way to accomplish this is to replicate the lookup data structure in multiple SRAM chips. This allows performance to scale to OC768 rates while still maintaining reasonably low cost.

References

[1] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," in *Proceedings of ACM SIGCOMM*, 1997.

[2] W. Eatherton, "Fast IP Lookup Using Tree Bitmap," *Washington University Master Thesis*, 1999.

[3] V. Srinivasan and G. Varghese, "Fast Address Lookups using Controlled Prefix Expansion," *ACM Transaction on Computer Systems*, vol. 17, 1999.

[4] B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *IEEE/ACM Transactions on Networking*, vol. 7, 1999.

[5] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," in *Proceedings of SIGCOMM*, 1997.

[6] F. Baboescu and G. Varghese, "Scalable Packet Classification," in *ACM Sigcomm*, San Diego, CA, Aug. 2001.

[7] J. Lunteren and T. Engbersen, "Fast and Scalable Packet Classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, May 2003.

[8] D. Taylor and J. Turner, "Scalable Packet Classification Using Distributed Crossproducting of Field Labels," in *Proceedings of IEEE Infocom*, 2005.

[9] G. Jacobson, "Succinct Static Data Structure," *Carnegie Mellon University Ph.D Thesis*, 1988.

[10] D. Taylor, J. Lockwood, T. Sproull, J. Turner, and D. Parlour, "Scalable IP Lookup for Programmable Routers," in *Proceedings of IEEE Infocom*, 2002.

[11] "BGP Reports," in *http://bgp.potaroo.net/*.

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

[13] M. Wang, S. Deering, T. Hain, and L. Dunn, "Non-Random Generator for IPv6 Tables," in *12th Annual IEEE Symposium on High Performance Interconnects*, Stanford University, CA, 2004.

[14] H. Liu, "Routing Table Compaction in Ternary CAM," *IEEE Micro*, vol. 22, 2002.

[15] K. Sklower, "A Tree-based Routing Table for Berkeley Unix," in *Winter Usenix Conference*, Dallas, TX, 1991.

[16] S. Sahni and K. S. Kim, "Efficient Construction of Multibit Tries for IP Lookup," *IEEE/ACM Transactions on Networking*, vol. 11, 2003.

[17] H. H.-Y. Tzeng, "Longest Prefix Search Using Compressed Trees," in *Proceedings of IEEE Global Communication Conference*, 1998.

[18] M. Desai, R. Gupta, A. Karandikar, K. Saxena, and V. Samant, "Reconfigurable Finite-State Machine Based IP Lookup Engine for High-Speed Router," *IEEE Journal on Selected Areas in Communications*, vol. 21, May 2003.

[19] R. J. Francis, J. Rose, and K. Chung, "Chortle: A Technology Mapping Program for Lookup Table-based Field Programmable Gate Arrays," in *27th Annual ACM/IEEE Conference on Design Automation*, 1991.

[20] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimaization in Lookup-Table Based FPGA Designs," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, 1992.