Scalable IP Lookups using Shape Graphs

Haoyu Song, Murali Kodialam, Fang Hao, T.V. Lakshman
Bell Labs, Alcatel-Lucent
{haoyusong, muralik, fangh, lakshman}@alcatel-lucent.com

***Abstract*—Recently, there has been much renewed interest in developing compact data structures for packet processing functions such as longest prefix-match for IP lookups. This has been motivated by several factors: (1) The advent of 100Gbps interfaces necessitating correspondingly fast packet processing algorithms with a compact memory footprint; (2) network virtualization leading to virtualization of physical router platforms making it critical to reduce high-speed memory needs per virtual router; (3) software routers built on multi-core processors requiring the use of compact data-structures that fit in on-chip caches for good performance.**

In this paper, we revisit this issue of developing compact data structures for key packet-processing functions. We develop a new data structure, called the shape graph, that significantly compacts the trie data-structure used for IP lookups. We accomplish this by identifying considerable structural similarities in IP lookup tries that have not previously been used in the literature for scalable IP lookups. We use these similarities to store lookup tries in a new graph data structure that has a significantly lower memory-footprint. Using real IP forwarding tables, we compare the memory usage of this new data structure to that of multi-bit tries and of Bloom filters used for IP lookups. The shape graph requires significantly less memory and allows the far more effective use of on-chip memory. This effective use of on-chip memory combined with multi-threading on a multi-core processor makes shape-graph-based IP lookups well suited for 100Gbps lookups. The small footprint also makes it well suited for use in router platforms that host a large number of virtual routers.

I. Introduction

There has been renewed research interest in developing memory-efficient data structures for high-speed packet processing. This is due to several driving factors that necessitate the use of low memory-footprint data structures. The expected transition to 100G interfaces requires the use of memory-efficient data structures, to achieve good performance while minimizing high-speed memory costs. Furthermore, the need to forward IPv6 packets and the growing forwarding table sizes increase the importance of achieving high-efficiencies in memory usage. Another important factor is the increasing use of multi-core processors for packet forwarding. In these systems, for achieving high-speeds, it is important to store entire forwarding data structures in on-chip caches. This necessitates the use of highly memory-efficient data structures. Finally, the growing use of virtualization implies that the same physical platform be able to act as multiple virtual routers. If, for isolation, each of the virtual routers maintains its own copy of forwarding data structures then memory-needs limit the number of virtual routers that can concurrently be hosted on a platform. To achieve good scaling in the number of virtual routers, it is important to maximize the memory-usage-efficiency of each virtual router. This makes it critical that memory-efficient data structures be used.

For virtual routers, one option for reducing memory usage is to use shared forwarding data structures with isolation being achieved by other mechanisms [9]. Our focus in this paper is on achieving high-memory efficiency for the case when each router maintains its own forwarding data structures. In particular, we focus on the problem of achieving significant memory-usage reduction for the longest prefix-match operation needed for IP lookups to determine the next-hop. Fast and efficient IP lookups has been well-studied and numerous algorithms have been proposed. Nevertheless, we find that it is possible to achieve significant reductions in memory usage. We consider the trie data structure, which is a fundamental data structure used for IP lookups. We find that there are significant structural, or shape, similarities between different part of a lookup trie. We exploit these similarities by using a new data structure that we call shape graphs. The shape graph permits operations similar to what can be done on trie but reduces structural redundancy, allowing us to greatly improve upon the memory-efficiency of tries. As with a multi-bit trie, a multi-bit shape graph allows the lookup process to examine multiple bits per memory access and boost the lookup throughput. The shape graph, however, scales much better in memory-usage with stride size than a multi-bit trie.

We also show that the shape-graph based method is more memory-efficient than the fast Bloom-filter based algorithms that have been recently proposed [7], [20]. With a multi-threading implementation using multiple on-chip memory blocks, the newly developed scheme can match the speed of Bloom-filter based methods while using less memory.

II. Shape Graphs: Exploiting Trie Similarity

Binary tries are a natural data structure for performing the longest prefix matches needed for IP lookups. The trie data structure is used to store the set of prefixes over which longest prefix matches must be done. For every valid prefix the corresponding next-hop information is also stored. Several variants of the binary trie, such as the multi-bit trie, have also been proposed for improving throughput and memory usage [6], [8], [18], [22], [23].

In recent work [21], we observed that at each level of a binary prefix trie, the number of isomorphic sub-trees is much smaller than the number of trie nodes. Consequently, if we count the total number of isomorphic sub-trees in the entire

978-1-4244-4634-6/09/$25.00 ©2009 IEEE

trie, it must be significantly smaller than the trie size (i.e. the total number of trie nodes).

We use an example to illustrate this point. Consider, as an example, a routing table as shown in Table I. The corresponding binary trie is shown in Figure 1. It contains eight nodes. We say two trees are isomorphic if by switching a node's left and right child nodes, along with the sub-trees they are rooted at, the two trees become identical. With this definition, in Figure 1, the single-node sub-trees rooted at d, g, and h are isomorphic because they are identical. Similarly, the sub-trees rooted at b, e, and f are also isomorphic. Note that the sub-tree rooted at e is isomorphic to the sub-tree rooted at f because if node f's left child is switched to the right, then the two sub-trees become identical. If we give each unique isomorphic sub-tree a unique *id* starting from 1 (The *id* 0 is intentionally reserved for a NULL shape) then, in Figure 1, there are only four unique isomorphic sub-trees. These are only 50% of the trie size.

TABLE I
SAMPLE PREFIX TABLE

prefix	next hop
*	P0
00*	P1
11*	P2
101*	P3
110*	P4

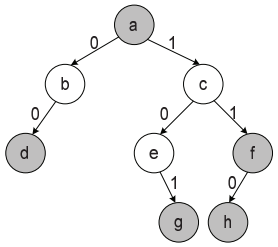

Fig. 1. The binary trie for the sample prefix table. The dark nodes stand for the valid prefixes

The difference between the two is much more dramatic when the tries are constructed from real IP prefix data. We tested the difference on a binary trie constructed from a snapshot of the BGP *AS1221* prefix table. The table has more than 210K IPv4 prefixes. The constructed binary trie has 576,534 nodes but has only only 48,762 unique isomorphic trees. This is only 8% of the trie size. Similar comparisons on many other prefix tables reveal the same dramatic differences between the number of nodes in the trie and the number of isomorphic sub-trees.

How can this observation, that the number of isomorphic sub-trees in a prefix trie is much smaller than the trie size, help us in reducing the memory consumption for IP lookups? The answer lies in the transformation of the trie data structure from a tree to a more compact graph where the compaction is accomplished using structural similarities in the trie. Assume the binary trie has n nodes and k isomorphic sub-tree groups. We first label each binary trie node with the *id* of the isomorphic sub-tree rooted at it. The resulting labeled trie, for our example, is shown in Figure 2(a). Now, we construct a directed graph with k vertices, where each vertex corresponds to one of the isomorphic sub-trees in the original trie. Edges between vertices are added in accordance with the relationship of the corresponding isomorphic sub-trees in the original trie. In our example, the trie node with label 4 has two child nodes with labels 2 and 3. Hence, in the constructed graph, vertex 4 has two directed edges pointing to vertices 2 and 3 as shown in Figure 2(b).

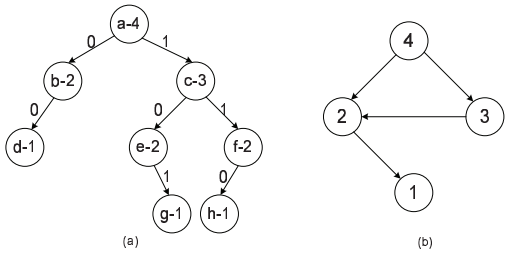

Fig. 2. Label the binary trie with isomorphic sub-tree ids and transform it to a directed graph

The graph resulting from the trie transformation has some interesting properties: (1) it has only one starting vertex which maps to the original trie root, and it has only one terminating vertex which maps to all the leaf trie nodes; (2) each vertex can have multiple incoming edges but can have at most two outgoing edges. Both these properties can be easily deduced from the transformation process. In fact, the graph is simply the result of condensing all the trie nodes with the same label to one graph node and removing the redundant trie branches. Since the graph consists of far fewer vertices and edges, it is not a surprise that it uses far less memory than the trie that it is based upon.

Unfortunately, this transformation results in too much loss of information that is inherent in the trie and hence makes the graph not yet practical for IP lookups. For example, one cannot perform the reverse transformation to recover the original trie. Neither can one walk the graph using a given IP address (to perform lookups) since the directed graph edges lose the notion of a '0' branch and a '1' branch that is inherent to the binary trie. However, a little change to the transformation process can easily solve these issues.

The change that we make to the transformation is that rather than grouping isomorphic sub-trees we instead group only identical sub-trees. The number of identical sub-tree groups is likely to be greater than the number of isomorphic sub-tree groups. However, note that as the number of identical sub-tree groups is smaller than the original trie, we still get reductions in memory usage compared to the trie. In the *AS1221* prefix table, the binary trie has 76,276 groups of identical sub-trees, as opposed to 48,762 groups of isomorphic sub-trees. Although the number of groups of identical sub-trees is 1.6 times larger than the number of groups of isomorphic sub-trees, it is still 7.6 times smaller than the number of trie nodes,

which implies a significant reduction in memory usage.

We name each unique sub-tree as a *shape*. We now re-label our example binary trie with the new shape *id*, as shown in Figure 3(a). We then construct the corresponding directed graph as shown in Figure 3(b). The graph contains five vertices, mapping to the five unique shapes in the trie. We term the graph constructed thus as a *shape graph*.

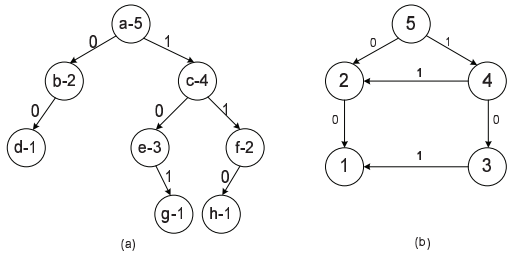

Fig. 3. Label the binary trie with shape ids and build the corresponding shape graph

The shape graph inherits all the properties of the graph built using isomorphic sub-trees. In addition, it preserves extra information from the trie allowing the graph to be used as a compact alternative for IP lookups. In the shape graph, each vertex (except for the terminating vertex) has exactly two ordered outgoing edges these map to the 0 and 1 branches of trie nodes. One can now recover the original trie from the graph alone. Also, one can use an IP address as input to walk the graph, and this walk will use the same number of steps to reach the terminating vertex as walking in the original binary trie to a leaf node would. This is easy to establish if we view the shape graph as being built by condensing together all the original trie nodes with the same label.

III. Efficient IP Lookups With Shape-Graphs

Next, we present our algorithm that uses the shape-graph data structure for fast memory-efficient IP lookups.

A. Counting the shape and labeling the trie

We first need to formalize the procedure to identify the shapes in a trie. We start with the following observation: Two binary trees are *identical* if and only if their left sub-trees are identical and their right sub-trees are also identical.

Since we label each node with the shape *id* of the sub-tree rooted at it, we can derive a simple corollary from the above observation:

Corollary 1: Two binary trees are identical if and only if their left child nodes have the same label and their right child nodes also have the same label.

In view of this corollary, we sketch the procedure to identify the shapes and to label the binary trie. The process only needs to do a post-order traversal of the trie (in one pass). For example, the trie in Figure 1 is traversed in the order d-b-g-e-h-f-c-a). All the leaf nodes are considered to be identical sub-trees and are assigned shape *id* 1. Therefore, we label all the leaf nodes with the number 1.

For all the other nodes, we examine the labels of their child nodes. If one child node does not exist, we assume that the non-existent child node bears a null shape *id* 0. Hence, for each node, we get a pair of labels (e.g. In Figure 3(a), trie node b gets a pair of labels $\{1, 0\}$). We use this pair of labels as the key to query a hash table: if the key is not in the hash table yet, we store the key along with the next unused shape *id* r in the hash table and then label the current trie node with number r; otherwise if the key is already in the hash table, we retrieve the value associated with the key and use it to label the current trie node. For our example, this process results the labeled trie exactly as shown in Figure 3(a). Since we assign the shape *id* consecutively, the label of the trie root tells us exactly how many unique shapes are in the trie.

B. Constructing The Shape Graph

After the trie is labeled, it is straightforward to construct the shape graph. We first allocate k vertices, where k equals the label of the trie root. Then the trie is traversed again in any order. For any visited node, if its label r has been seen before, nothing needs to be done. If the label r has not been seen yet, we retrieve the label of its '0'-branch child node, s, and the label of its '1'-branch child node, t (note that the corresponding label is 0 for a non-existent child node). In the shape graph, we set a directed '0'-edge from vertex r to vertex s and a directed '1'-edge from vertex r to vertex t. Note that if s equals t, then the walk from vertex r always goes to the same next vertex s, no matter what the input is.

To accelerate the process, we can terminate the trie traversal as soon as all the k shape *id*s have been accessed. We observe that most of the same shaped sub-trees are deep down in the trie, so a breadth-first traversal order typically yields the shortest processing times.

C. Leaf pushing

Now we can use the shape graph as a compressed form of the trie and use if for IP lookups. However, there is another issue that must be addressed. In the shape graph, it is possible for a vertex to be reachable through multiple paths. Some paths leading to a vertex may indicate valid prefixes while others may not. For example, in Figure 3(b), vertex 2 actually maps to paths "0*" and "11*", in which "11*" is a valid prefix but "0*" is not. Clearly, for IP lookup we need to differentiate between these two cases of valid and invalid prefixes. How can we do this differentiation? One can explicitly store the valid prefix with the vertex for comparison, but this will increase storage costs. Recall that one nice feature about the binary trie is that any node can be reached through only one path, so if a node is indicated as a valid prefix node, the unique path leading to it implies the prefix. No extra information needs to be stored by the trie node. This is a great advantageous to achieving savings on memory needs.

We want to preserve this nice trie feature, of eliminating ambiguities in prefix validity, in the shape graph as well. The key to do this is based on observing that if there is only a single vertex in the shape graph that indicates all the valid

prefixes, then if we reach this vertex, we know that we have matched a valid prefix and the matched prefix is implied by the walking path. This special vertex is the terminating vertex, where all the paths converge. Since the terminating vertex actually maps to all the leaf nodes in the original trie, to have this feature in the shape graph, we require that only the leaf trie nodes in the original trie can indicate valid prefixes. This is achieved by applying a simple and widely used technique, leaf pushing [23], on the binary trie before constructing the shape graph.

Leaf pushing first grows the trie to a full tree (i.e. all the non-leaf nodes have two child nodes) and then pushes all the prefixes to the leaf nodes. For our example, the leaf-pushing trie is shown in Figure 4(a) and the updated prefix table after leaf pushing is shown in Table II. We can see that leaf pushing has the negative effect of expanding the prefix table size as well as the corresponding trie size. For the previously considered *AS1221* table, after leaf pushing, the prefix table expands 1.7 times and the corresponding trie also expands 1.3 times. However, on the positive side, leaf pushing actually results in fewer shapes. Consequently, the corresponding shape graph is smaller (e.g. the shape graph in Figure 4(b) contains one less vertex than before). For the *AS1221* table, the shape graph after leaf pushing contains 51,962 vertices, a 32% reduction compared to the shape graph before leaf pushing. The reason for this is that the leaf pushing trie looks more regular and therefore results in more identical sub-trees.

This size reduction of the shape graph helps offset the impact of the prefix table expansion. Note that, leaf pushing and another related technique (prefix expansion) have been widely used in many high performance IP lookup algorithms [7], [9], [23]. We will compare our scheme against these later.

TABLE II
EXPANDED PREFIX TABLE AFTER LEAF PUSHING

prefix	next hop
00*	P1
01*	P0
100*	P0
101*	P3
110*	P4
111*	P2

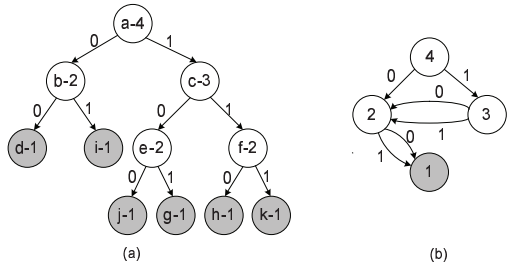

Fig. 4. Label the leaf-pushing trie with shape ids and build the shape graph

The most important consequence of leaf pushing is that (1) only the terminating vertex maps to valid prefixes, (2) the prefix is implied by the walking path in the shape graph, and (3) each IP address used as input for walking the shape graph can match only one prefix which is the longest match. Therefore, all next-hop information can be stored in one hash table associated with the terminating vertex.

Although, at this point, we have the needed memory-efficient data structure for IP lookups, we describe some further optimizations before presenting the actual lookup algorithm.

D. Multi-bit Shape Graphs

Although the shape graph is smaller in size, the lookup throughput using the shape graph obtained from a binary trie is no better than using the binary trie. When tries are used for IP lookups, one generally resorts to using a multi-bit trie for improved throughput [8], [22], [23]. A multi-bit trie with a stride of s can boost the throughput roughly by a factor of s. A multi-bit trie can also reduce the total number of trie nodes. However, the overall memory consumption increases rapidly as the stride size increases because the node size grows exponentially with the stride size. This increase in node size significantly outpaces the reduction in the number of trie nodes. Because of this, the stride cannot be set to be too large. The multi-bit trie with a stride of two for our example prefix table is shown in Figure 5.

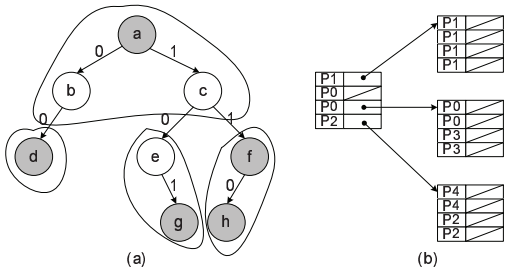

Fig. 5. Multi-bit trie and the data structure with the stride of 2

We can construct the multi-bit shape graph in a manner similar to the construction of binary shape graph. The multi-bit shape graph is derived from the binary shape graph. Initially, the new multi-bit graph contains only one starting vertex. With a stride of size s, we walk the original shape graph using each of the 2^s s-bit patterns from the starting vertex. We add all the finally reached vertices in the new graph, if they are not already in it. The vertices are connected with edges, with each representing a stride for a different s-bit pattern. We repeat the previous step for each of the newly added vertices The process terminates when no vertex can reach other vertices other than the terminating vertex.

Figure 6(b) and 6(c) show a 2-bit shape graph and 3-bit shape graph respectively, for our example prefix table. From the figures, we can see that with the stride increasing, the graph size shrinks. At the same time, the edges out of each vertex increases. However, our evaluation (see Section V) shows that the overall memory consumption increases at a much slower

pace than for the multi-bit trie. Hence the multi-bit shape graph is more scalable.

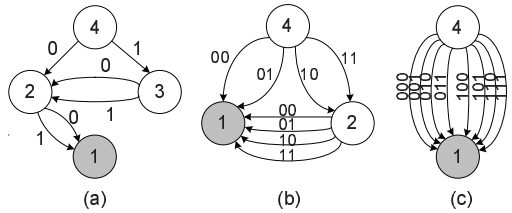

Fig. 6. Multi-bit shape graph with stride of 1, 2, and 3

E. Avoiding prefix expansion

Now we tackle another issue related to memory consumption. As we can see in Figure 5, multi-bit trie requires prefix expansion in each node. All the prefixes are expanded to the closest length l, where l is a multiple of stride s. This is another source leading to the memory inefficiency. The multi-bit shape graph faces the similar problem. For example, in the 3-bit shape graph shown in Figure 6(c), the edges "000" and "001" are actually from just one original prefix "00*". But the terminating vertex cannot tell this information so we have to split the original prefix "00*" into two individual prefixes "000" and "001". With a small stride, such prefix expansion is still tolerable. But when the stride is large, the expanded prefix table would become quite large. Such expansion is especially problematic for the shape graph, because unlike the multi-bit trie where the next hop can be embedded into each trie node, the shape graph uses a hash table to store all the $\{prefix, nexthop\}$ pairs.

We propose a simple technique to avoid prefix expansion, although this will slightly increase the size of each graph vertex. Since each vertex (except the terminating vertex) has 2^s outgoing edges, we maintain a 2^s-bit bitmap in each vertex to indicate which group of edges are for the same original prefix, if they lead to the terminating vertex. This is possible because a prefix always expands to a set of prefixes with consecutive values. We can use this bitmap to infer the actual length of the prefix so as to avoid prefix expansion.

If the prefix length is l, the last walk step contains only $r = l \mod s$ bits, which will expand to 2^{s-r} consecutive edges. The process of setting the bitmap for a vertex is as follows: we use a temporary binary flag which can be either 0 and 1. The flag is initialized to 0. Starting from the first edge, if the next t edges lead to the terminating vertex and they belong to the same original prefix, we set the corresponding t bits in the bitmap to the current value of the flag. We then flip the flag after setting t bits. Then we look at the next edges and continue the process. If an edge does not lead to the terminating vertex, we set the corresponding bit in the bitmap to the current value of the flag, and flip the flag. The process stops when all the edges have been scanned and all the bits in the bitmap have been set. As a result, the generated bitmap contains consecutive 0's and 1's. A string of two or more same-value bits implies the corresponding edges belong to the same prefix. For example, in Figure 6(c), the bitmap in vertex 4 should be "00110101". The first two bits "00" correspond to prefix $00*$; the next two bits "11" correspond to prefix $01*$, and so on.

For the lookup process, this bitmap is used only if the next vertex is the terminating vertex; otherwise it is ignored. Suppose when we look up a prefix in the graph, we have traversed k steps and find the next edge leads to the terminating vertex. If the corresponding bit for this edge belongs to a string of t ($1 \leq t \leq 2^{s-1}$) consecutive 1s (or 0s) in the bitmap, then the prefix length can be calculated as $s(k+1) - \log_2 t$. In the above example, suppose we look up an address "010". Starting at vertex 4, we find this corresponds to the third outgoing edge, which leads to the terminating vertex 1. We then find the third bit in the bitmap of vertex 4 is "1", which belongs to a string of two consecutive 1s. From the above equation, the prefix length is $3 \times (0+1) - \log_2 2 = 2$ and therefore the best matching prefix is $01*$.

The relatively insignificant overhead of 2^s-bit per vertex makes it possible to use large stride to increase lookup throughput without major increase in memory.

F. Performing IP Lookups

The IP lookup process should be fairly straightforward at this point. We use the given IP address as input to walk the graph, in the way similar to walking a trie. In the last step before entering the terminating vertex, we use the vertex bitmap plus the walk steps to calculate the length of the best matching prefix, as explained in the previous section. Finally, when we reach the terminating vertex, we use the matching prefix as key to retrieve the next hop information from a hash table.

G. Incremental update

IP forwarding table may be frequently changed over the time due to temporal route fluctuation. Therefore it is necessary for a successful IP lookup algorithm to support fast incremental updates. We simply cannot afford to rebuild and reload the entire data structure for each update.

Incremental updates include change of next hop and IP prefix insertions and deletions. Change of next hop is relatively easy since we just need to modify the hash table entry for the corresponding prefix.

To handle prefix insertions and deletions, the route processing software works on the leaf-pushing trie first and then modifies the shape graph and the hash table if necessary. A prefix deletion can be done without actually removing any trie node, hence the shape graph can remain intact and we only need to update the next-hop associated with the deleted prefix in the hash table. For example, if $\{110*, P4\}$ is deleted in Table I, we only need to update the next hop of prefix "110*" to $P2$ in the hash table. Such "lazy" deletion can simplify the operation and save memory accesses. It may also benefit later insertions. For example, when a prefix is frequently deleted and inserted during route fluctuation, we only need to modify

the hash table without touching the shape graph for both operations.

The prefix insertion is a bit more complicated. In the following discussion, we first use binary shape graph as an example. If insertion of a new prefix does not create any new node in the trie, the shape graph will not change, and hence only the hash table needs to be updated. However, when the insertion creates some new nodes in the trie, the shape graph needs to be updated accordingly in addition to hash table updates. The number of memory accesses for the update is upper-bounded by the trie depth. This is because the newly added prefix can only alter the shapes of all its ancestor nodes in the trie. If a node's shape *id* is altered to another existing shape *id*, no further modification to the shape graph is necessary because the connectivity between vertices for the existing shapes has already been set up. On the other hand, if a node's shape is altered to a new shape, we need to update the node with a new shape *id*. At the same time, we generate a new vertex in the shape graph and add two outgoing edges from the new vertex to the two vertices representing its two child shapes. We repeat this process bottom up till we reach the trie root. If the trie root's shape *id* is new, we generate a new vertex having this shape *id* in the shape graph and make it the new starting vertex.

The prefix insertion and deletion can yield some vertices unused. The unused vertices should be recycled regularly (i.e. remove the unused vertices from the memory and reuse the corresponding shape ids for new shapes) to avoid exhausting the memory if otherwise unattended. This can be easily achieved in the processing software by tracking the usage of each shape id.

Figure 7 shows what happens when new prefix "001*", "1100*", and "1111*" are inserted one by one into shape graph in Figure 4. Both the resulting trie and the corresponding shape graph are shown in the figure. After prefix "001*" is inserted, the old starting vertex 4 becomes redundant because it is not used by any lookup. However, we do not need to remove it from the graph immediately because later on it may be used again. As shown in Figure 7, after prefix "1111*" is inserted, vertex 4 is reused as an internal vertex.

For a multi-bit shape graph, one incremental update requires at most $\lceil \frac{d+1}{s} \rceil$ memory accesses, where d is the binary trie depth and s is the stride.

Note that the actual memory operation for the shape graph updates is just memory write. When an update involves a series of memory writes, we do not activate the new starting vertex until all the memory writes are done. During this period, there is no need to block the normal lookups. They still start from the old starting vertex until the update is finished. When these write requests are interwoven with the normal memory accesses in the same pipeline, the update does not affect the lookup correctness. The infrequent update also has little impact to the lookup throughput. For instance, each update to a shape graph with a stride of 6 needs at most 6 memory writes. If there is an update every 1ms, there will be 6K memory writes per second, which consumes only 0.003% bandwidth of a moderate 200MHz memory.

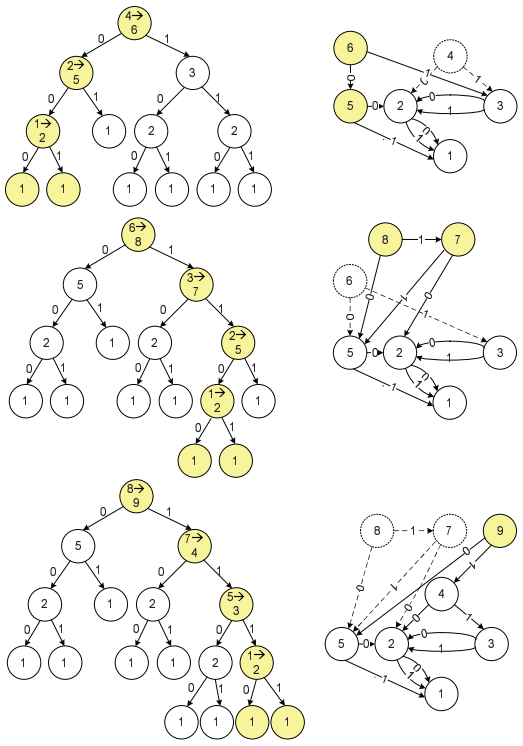

Fig. 7. Illustration of the shape graph updates as the new prefixes "001*", "1100*", and "1111*" are inserted sequentially. Left side is the updated leaf-pushing binary trie and the right side is the updated shape graph. The dark nodes (vertices) indicate the newly inserted or updated nodes (vertices). The dashed vertices and edges in the shape graph indicate the unused vertices and edges after the update

IV. Implementation Considerations

A. Fast shape graph lookup

The memory consumed by the shape graph is small enough to fit in the on-chip block memory in today's ASICs, FPGAs, and Network Processors. This allows us to deploy multiple memory blocks and spread the graph vertices into them to improve the lookup throughput. This level of parallelism is similar to that of the super-scalar pipeline architecture. Multiple packets are dispatched to conduct lookups simultaneously. Each memory access retrieves a pointer giving the block id and block offset for the next access. This process repeats until the terminating vertex is reached. The matching prefix is then derived and used to search the off-chip hash table and get the next-hop.

Such an architecture is illustrated in Figure 8. The use of multiple memory blocks significantly increases the aggregated bandwidth. Since each packet may need a different number of memory accesses to finish the lookup, the final lookup results may appear out of order. We arrange a reordering buffer at the output side so that the lookup results output in the order that the packets are fed into the search engine.

Ideally, each memory block should roughly contain the same number of vertices so the memory block size can be

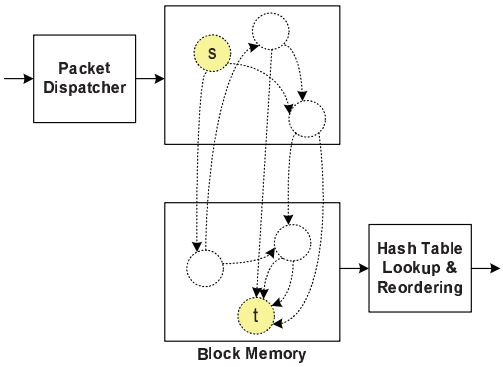

Fig. 8. Distribute graph vertices into multiple memory blocks to improve the lookup throughput. s is the starting vertex and t is the terminating vertex

equalized for easier engineering and better memory efficiency. In addition, the vertices should be spread across different memory blocks in a way that the lookup accesses to each memory block is equalized in order to maximize throughput, or minimize the number of outstanding packets that are needed to fill up the bandwidth. This problem is similar to the classical bin packing problem where one tries to pack a set of variable sized items into the minimum number of fixed capacity bins. The difference is that in our problem we have a fixed number of bins (i.e. memory blocks) with unlimited capacity and we need to satisfy the packing goals as discussed above.

We assume each graph vertex is associated with a weight which indicates the probability for it to be accessed by a prefix lookup. We use the min-max heuristic [14] to assign each vertex to one of the k memory blocks: we sort the vertices in decreasing weight order and then assign each vertex in the current least weighted memory block according to the weight order; we repeat the process until all vertices are assigned. Note that we leave out the termination and starting prefixes due to the following reasons. We actually never need to access the terminating vertex. The matching prefix can be determined at the vertex one step before the terminating vertex. The starting vertex is the most weighted vertices which are accessed by every packet lookup. However, it can be handled in the control logic rather than in the memory to save the memory bandwidth consumption.

We can assign the weight to the vertices in one of two ways. In the static method, we assume each prefix in the table is accessed with the same frequency. The weight of all the vertices is initialized to 0. Then for each prefix, we increment the weight of the accessed vertices on the prefix walk path by one. As a result, a vertex with weight w means there are w prefixes passing through it.

In practice, the above assumption is not always true. Prefix access can be fairly imbalanced. It is possible that majority of the lookups are concentrated on a small subset of prefixes. The dynamic method therefore keeps track the access rate of each vertex and update their weight accordingly. When a better vertex distribution is preferred, we run the algorithm and escort the affected vertices to their new host memory blocks. The dynamic method can help balance the accesses to the memory blocks to the full extent.

The max-min heuristic works extremely well for shape graph with static assigned weight. For the *AS1221* prefix table, assuming 4 or 8 memory blocks are used, the maximum deviation of the overall weight in any memory block is less than 10 from the average (reflecting the bandwidth balance) and the maximum deviation of the number of vertices in any memory block is at most a few tens from the average (reflecting the memory size balance), which are negligibly small.

In theory, with the perfect vertex distribution, $\lceil \frac{d+1}{s} \rceil - 2$ memory blocks are enough to support finishing a lookup in just one clock cycle in the worst case, where d is the length of the longest prefix (e.g. 32 in the case of IPv4) and s is the stride. The deduction of 2 in the equation is because both starting and terminating vertex are not actually stored in the memory.

This means four memory blocks are sufficient for stride size of six. However, to tolerate the temporal access imbalance, we can use more memory blocks. Note that this will not increase the memory consumptions, since each memory block will hold fewer vertices proportionally.

B. *Efficient hash table construction*

Performance of the hash table has direct impact on both storage and lookup throughput. There are many ways to implement the hash table efficiently with compact storage and low collision probability [15], [19]. In this paper we use a simple yet efficient multiple hashing scheme proposed in [3]. In this scheme, there are k independent hash functions and each table bucket has n slots to hold up to n $\{prefix, nexthop\}$ pairs. Each prefix is hashed k times into k candidate buckets, but it is only stored in the lightest loaded one. Consequently, a prefix lookup needs to access the hash table k times using the k hash functions. All prefixes stored in the k accessed buckets need to be compared to find the match. By using multi-port memory or multiple parallel memory modules, such memory accesses can also be parallelized.

With the fine tuned parameters, the hash table is actually quite compact and has extremely low overflow rate. In the rare case that a prefix cannot find an empty slot to store in the selected buckets, it is treated as an exception and stored in a very small on-chip TCAMs. This is a common technique used by many hashing-based algorithms [7], [20].

For example, for the *AS1221* table after leaf pushing, we can use a hash table with the number of buckets as half of the number of prefixes and set $k = 4$ and $n = 3$. There are typically less than 10 overflow prefixes for many different trials each with different hash functions. If we assume each $\{prefix, next\ hop\}$ pair uses five bytes (4-byte prefix plus 1-byte next hop), then on the average a prefix consumes 60-bit memory. The total memory consumed by the hash table is therefore 21.5Mb.

Some recent proposals build efficient Bloom filter based data structure that can directly return the *id* of the group that

an element belongs to [4], [5], [10], [16]. Instead of hash table, we can also combine the shape graph approach with such Bloom filter data structures for IP prefix lookup. Given an IP address, the shape graph lookup returns the best matching prefix. We can group and store the prefixes according to their next-hop output port such that the lookup will directly return us the output port *id*. Although this kind of data structure has the potential to significantly reduce the memory consumption, in this paper we still use the aforementioned hash table as the basis for conservative performance evaluation.

V. Performance Evaluation

For the performance evaluation, we choose two representative real-world prefix tables: a large table *AS1221* that contains 215,454 prefixes, and a small table *Maewest* that contains 27,930 prefixes. Both numbers are before leaf pushing.

A. Comparison with multi-bit trie algorithm

We first compare the shape graph algorithm with the multi-bit trie algorithm. As shown in Figure 9, the shape graph contains significantly fewer number of nodes than the multi-bit trie for all different strides.

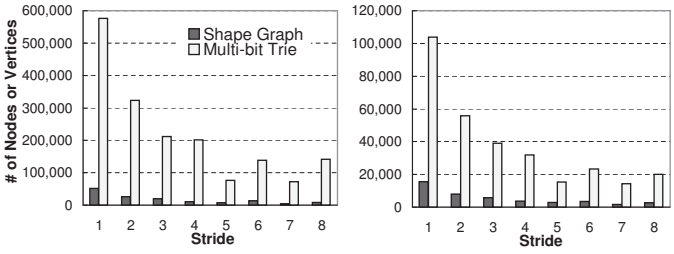

Fig. 9. Number of trie nodes versus number of graph vertices. The left side is for AS1221 and the right side is for Maewest

One can argue that the multi-bit trie does not need an extra hash table so a fair comparison should consider the overall memory consumption instead. Moreover, some algorithms such as Lulea [6] and Tree Bitmap [8] encodes the multi-bit trie cleverly so the memory consumption is significantly reduced. To see where our algorithm stands, the comparison of the multi-bit trie algorithm, the Tree Bitmap algorithm, and ours is as follows. We assume the next-hop information consumes 8 bits and each pointer consumes $\lceil \log_2 m \rceil$ bits, where m is the number of nodes in the trie or vertices in the graph. So with the stride of s, the memory consumed by the multi-bit trie algorithm is $m2^s(8 + \lceil \log_2 m \rceil)$ and the memory consumed by the Tree Bitmap algorithm is $m(2^s + 2^s - 1 + \lceil \log_2 m \rceil + \lceil \log_2 n \rceil) + 8n$. Likewise, the shape graph algorithm consumes $m2^s(1+\lceil \log_2 m \rceil)+8n'$ bits. The extra 2^s bit in the above formulation is for the bitmap as discussed in Section III-E. We assume a perfect hashing implementation. n' is the number of prefixes after leaf pushing.

To compare the scalability of the algorithms, we normalize the memory consumption as the average number of bits consumed by each original prefix. The comparison of the three algorithms on the two prefix tables are shown in Figure 10. The shape graph algorithm shows clear advantage for both cases. Although the Tree Bitmap algorithm closely tracks the memory efficiency of the shape graph algorithm, it suffers from significant incremental update cost [8]. Finally, we note that the memory consumption is not monotonically increasing with the stride. This is because sometimes the increasing of the stride can significanlty reduce the number of trie nodes. Even the trie node size increases, the overall memory consumption still appears decreasing at these settings.

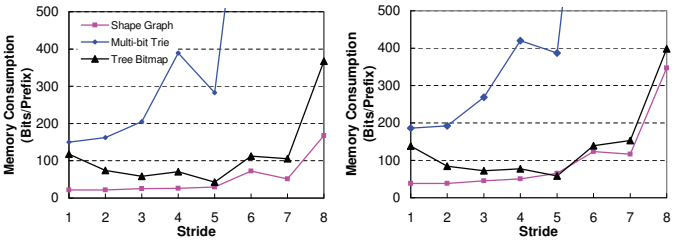

Fig. 10. Overall memory consumption comparison on two prefix tables. The left side is for AS1221 and the right side is for Maewest

B. Comparison with Bloom filter based algorithm

We also compare our algorithm with the Bloom filter based IP lookup algorithm proposed in [7]. Since both algorithms use a hash table for the next hop lookup, we assume they use the same hash table implementation and hence consume the same amount of memory. Therefore we focus on comparison of the memory sizes of the shape graph and the Bloom filter.

According to the Bloom filter theory, if we want to keep the false positive rate below 1×10^{-5}, an item will consume at least 24 bits in the best case [2]. Therefore 5-Mb SRAM bits are needed to construct the Bloom Filter for the *AS1221* table,

On the other hand, the shape graph for *AS1221* consumes less than 16-bits per prefix for a stride up to 5. Furthermore, if we allocate more memory blocks for the shape graph as proposed in IV-A, the total memory consumption will not change, but the aggregate throughput will become as good as or even better than the Bloom filter based algorithm.

Figure 11 shows more results for the *AS1221* table. With eight bits per prefix allocated, the Bloom filter based algorithm exhibits a poor false positive rate ($> 2\%$), while the shape graph can still support a stride of 2. More importantly, shape graph can reach very high throughput even for small stride as long as enough memory blocks can be used.

In the Bloom filter based algorithm, to avoid using too many Bloom filters, prefix expansion is often applied to reduce the number of unique prefix lengths [7]. While this can reduce the number of memory blocks, each memory block must increases its size to handle more expanded prefixes. Also note that shape graph algorithm always give accurate results, while the false positive generated by the Bloom filter can negatively impact the throughput performance.

The Bloom filter based algorithm also relies on multi-port memory blocks to achieve the desired throughput. This is

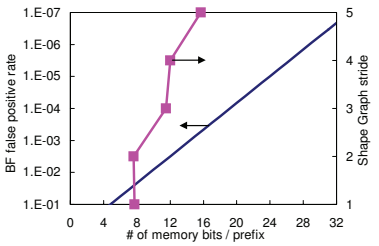

Fig. 11. On-chip memory consumption for different Bloom filter false positive rates and different shape graph strides. the memory consumption is normalized to bits per prefix.

because each Bloom filter contains multiple hash functions and all of them need to access the memory. However, majority devices provide embedded memory blocks with at most two ports. While it is possible to use dual-port memory blocks to construct the Bloom filters, it can significantly complicate the circuit design [7]. On the contrary, in our algorithm implementation, each packet needs only to access each memory block one time or less on the average. So the single port memory is sufficient and the dual-port memory can effectively double the lookup throughput.

In the following evaluation, we assume both algorithms use k memory blocks. For the shape graph algorithm, we use the equation $k = \lceil \frac{d+1}{s} \rceil - 2$ to determine the stride s, so the algorithm can finish one shape graph lookup per clock cycle in the best case. For the Bloom filter based algorithm, k memory blocks can support k Bloom filters to handle k different prefix lengths. The prefixes with lengths that are not covered by the Bloom filters need to be expanded to multiple longer prefixes. We assume each of the covered length is equal to some multiple of $\lceil d/k \rceil$ (e.g. given $k = 8$, the set of supported prefix lengths is 4, 8, 12, 16, 20, 24, 28, and 32). Figure 12 shows the memory consumption normalized as bits per original prefix for both algorithms. Performance of the Bloom filter based algorithm are shown in two different false positive settings. We observe that the memory consumption of our algorithm is roughly equal to the that of the Bloom filter based algorithm under the false positive rate of 0.001. The expanded prefix table also needs a larger off-chip hash table. Since our algorithm avoids the prefix expansion, it does not incur such extra overhead.

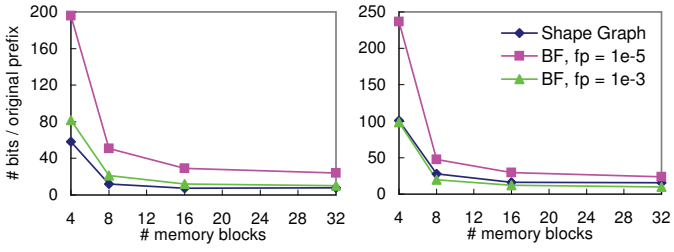

Fig. 12. Memory consumption for different number of memory blocks. The left side is for the AS1221 table and the right side is for the Maewest table.

It is possible to further optimize the set of prefix length thresholds by using controlled prefix expansion [23]. But regardless of which prefix lengths to use, the prefix table is still significantly expanded. As we show earlier, a prefix consumes 24 bits to achieve the $1e-5$ false positive rate even without any prefix expansion. Our algorithm consistently uses smaller memory when more than eight memory blocks are available.

C. Experiments on IPv6 tables

It is a little difficult to evaluate the algorithm for IPv6 since there are no large real-world IPv6 forwarding tables available today. We first use a real IPv6 BGP table that contains about 900 prefixes. Prefix length in this table ranges from 16 to 64.

We also synthesize a large IPv6 forwarding table using the methodology developed in [25]. The authors observe that while it is difficult to predict the structure of future large scale IPv6 forwarding lookup tables, it is possible to use the IPv6 address allocation schemes and the characteristics of current IPv4 tables to infer information that can be used to generate realistic IPv6 tables. In our experiment, we generate an IPv6 table based on the *AS1221* IPv4 table. The table contains 215,518 prefixes.

The comparison between the shape graph and the multi-bit trie is shown in Figure 13. We can see that the trie for IPv6 contains many more nodes than that for IPv4 with the same number of prefixes. Therefore, the data structure scalability is more critical for the IPv6 case. We observe that the shape graph is much smaller and scales better than the multi-bit trie for all the strides.

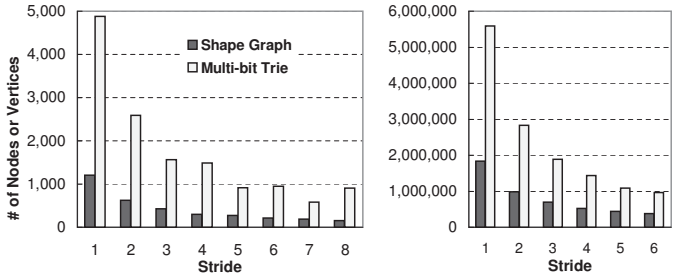

Fig. 13. The number of shape graph vertices vs. the number of multi-bit trie nodes for IPv6 forwarding tables. The left side is for the IPv6 BGP table and the right side is for the synthesized IPv6 table

IPv6 poses extra challenges to the Bloom filter based algorithm due to the large number of unique prefix lengths. Limiting the number of Bloom filters would cause a much larger prefix expansion factor. Hence the shape graph algorithm outperforms the Bloom filter based algorithm even more significantly in IPv6 compared to IPv4.

D. Throughput analysis

100GbE is being standardized and the router with 100GbE line cards needs to process 150 million packets per second per port in the worst case. The current FPGAs, ASICs, and memory components can comfortably work at 300MHz, which means at least two clock cycles are available to finish one forwarding lookup decision. The architecture proposed in Section IV can easily sustain such throughput and has the potential for even higher line speed.

VI. Related Work

IP route lookup is a well-studied problem. Although TCAM is handy and guarantees the high throughput for IP lookups, it suffers from the excessive power consumption, high cost, and low density. Therefore, the algorithmic solutions are still very popular alternatives.

High performance algorithms are often implemented in hardware as dedicated search engines involving control components and memory components. The most popular IP lookup algorithms are trie-based [6], [8], [18], [22], [23]. While each generation of the trie-based algorithm becomes more memory efficient and allows faster lookups, the performance still decreases linearly as the tree depth increases, making these algorithms less suitable for the emerging IPv6 lookups.

Another approach for fast IP lookups is to use memory pipelines [1], [11]–[13]. A deep pipeline can be used to produce one lookup result every clock cycle. However, the high aggregated memory bandwidth needed by all the pipeline stages prohibits it to be implemented with commodity memory devices. Using dedicated lookup engine with embedded memory for the pipeline stages requires the data structure to be small since the on-chip memory is still costly.

Hashing is also used for IP lookups [17], [24]. The algorithm described in [7] utilizes the on-chip memory blocks to construct multiple Bloom filters to parallelize the hashing process, therefore the lookup throughput is greatly enhanced. Our comparison shows that our algorithm is superior to it in terms of memory efficiency.

Virtual routers call for better data structure scalability to the algorithm design. [9] describes a simple scheme to insert all the prefixes belonging to different virtual routers into one trie. Orthogonal to this approach, our algorithm seeks to compress the lookup data structure of a single virtual router so the overall router scalability can be fulfilled.

VII. Conclusion

Technology advancement such as faster line cards, network virtualization, and software routers have generated renewed interest in developing compact data structures for packet processing. IP lookup algorithms need to support the line-speed forwarding without incurring high system cost and power consumption. For example, 100GbE line card requires the search engine to finish 150 million packet lookups per second. Few algorithms can reach such high throughput so far. We believe in order to meet such demands, an algorithm need to be optimized at the architectural level which combines multiple components and takes fully advantages of the hardware parallelism that the system offers.

In this paper, we have exploited one of the intrinsic characteristics of the IP forwarding table to build a very compact lookup data structure that scales to very large forwarding tables. When implemented properly using multiple parallel memory blocks, the algorithm can sustains extremely high lookup throughput. We show through real world IP forwarding tables that our algorithm outperforms both the multi-bit trie algorithm and the Bloom filter based algorithm in both memory size and lookup throughput. The algorithm can be implemented in FPGAs and ASICs, as well as the multi-core processors. For IP lookups on the BGP table with more than 200K prefixes, with a memory consumption as low as about 100 bits per prefix (including on-chip and off-chip memory), the algorithm can support line-speed lookup for the 100Gbps line speed.

Our future work includes implementing the algorithm in hardware and evaluating its performance using real Internet traffic.

References

[1] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A Tree Based Router Search Engine Architecture with Single Port. In *ISCA*, 2005.

[2] B. Bloom. Space/Time Trade-offs in Hash Coding With Allowable Errors. *Communications of the ACM*, July 1970.

[3] A. Broder and M. Mitzenmacher. Using Multiple Hash Functions to Improve IP Lookups. In *IEEE INFOCOM*, 2001.

[4] F. Chang, K. Li, and W. chang Feng. Approximate caches for packet classification. In *IEEE INFOCOM*, 2004.

[5] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[6] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *ACM SIGCOMM*, 1997.

[7] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest Prefix Matching using Bloom Filters. In *ACM SIGCOMM*, 2003.

[8] W. Eatherton, G. Varghese, and Z. Dittia. Tree Bitmap: hardware/software IP Lookups with Incremental Updates. *ACM SIGCOMM Computer Communication Review*, 2004.

[9] J. Fu and J. Rexford. Efficient IP Address Lookup with a Shared Forwarding Table for Multiple Virtual Routers. In *ACM CoNEXT*, 2008.

[10] F. Hao, M. Kodialam, T. V. Lakshman, and H. Song. Fast Multiset Membership Testing Using Combinatorial Bloom Filters. In *IEEE INFOCOM*, 2009.

[11] J. Hasan and T. N. Vijaykumar. Dynamic Pipelining: Making IP Lookup Truly Scalable. In *ACM SIGCOMM*, 2005.

[12] W. Jiang and V. K. Prasanna. Beyond TCAMs: An SRAM-based Multi-Pipeline Architecture for Terabit IP Lookup. In *IEEE INFOCOM*, 2008.

[13] S. Kumar, M. Becchi, P. Crowley, and J. S. Turner. CAMP: Fast and Efficient IP Lookup Architecture. In *ACM/IEEE ANCS*, 2006.

[14] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *ACM SIGCOMM*, 2006.

[15] S. Kumar, J. Turner, and P. Crowley. Peacock Hash: Fast and Updatable Hashing for High Performance Packet Processing Algorithms. In *IEEE INFOCOM*, 2008.

[16] Y. Lu, B. Prabhakar, and F. Bonomi. Bloom filters: Design innovations and novel applications. *Allerton Conference*, 2005.

[17] J. V. Lunteren. Searching Very Large Routing Tables in Wide Embedded Memory. In *IEEE Globecom*, 2001.

[18] S. Nilsson and G. Karlsson. IP Address Lookup using LC-Tries. *IEEE Journal on Selected Areas in Communications*, June 1999.

[19] H. Song, S. Dharmapurikar, J. S. Turner, and J. W. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: an Aid to Network Processing. In *ACM SIGCOMM*, 2005.

[20] H. Song, F. Hao, M. Kodialam, and T. Lakshman. IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards. In *IEEE INFOCOM*, 2009.

[21] H. Song, M. Kodialam, F. Hao, and T. Lakshman. Building Scalable Virtual Routers with Trie Braiding. In *unpublished manuscript*, 2009.

[22] H. Song, J. Turner, and J. Lockwood. Shape Shifting Tries for Faster IP Lookup. In *IEEE ICNP*, 2005.

[23] V. Srinivasan and G. Varghese. Faster IP Lookups Using Controlled Prefix Expansion. In *ACM SIGMETRICS*, 1998.

[24] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *ACM SIGCOMM*, 1997.

[25] M. Wang, S. Deering, T. Hain, and L. Dunn. Non-random Generator for IPv6 Tables. *12th IEEE HotInterconnects*, 2004.