A Novel Scalable IPv6 Lookup Scheme Using Compressed Pipelined Tries

Michel Hanna, Sangyeun Cho, and Rami Melhem

Computer Science Depratment
University of Pittsburgh
Pittsburgh, PA, 15260, USA
{mhanna,cho,melhem}@cs.pitt.edu

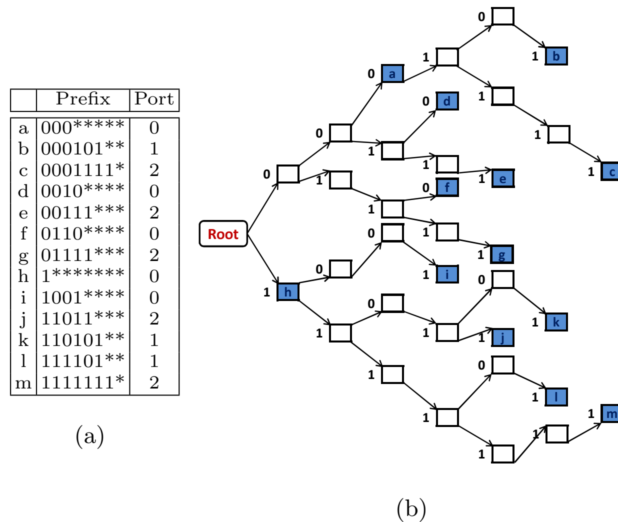
Abstract. An IP router has to match each incoming packet's IP destination address against all stored prefixes in its forwarding table. This task is increasingly more challenging as the routers have to: not only keep up with the ultra-high link speeds, but also be ready to switch to the 128-bit IPv6 address space while the number of prefixes grows quickly. Commercially, many routers employ Ternary Content Addressable Memory (TCAM) to facilitate fast IP lookup. However, TCAMs are power-eager, expensive, and not scalable. We advocate in this paper to keep the forwarding table in trie data structures that are accessed in a pipeline manner. Especially, we propose a new scalable IPv6 forwarding engine based on a multibit trie architecture that can achieve a throughput of 3.1 Tera bits per second.

Keywords: IPv6, Tries Compression, Next Generation Internet

1 Introduction

In the IP lookup (or forwarding) problem, the router has to match the destination address of every incoming packet against its forwarding table to determine the packet's next hop on its way to the final destination [22]. An entry in the forwarding table, or an IP prefix, is a binary string of a certain length followed by wild card (don't care) bits and an output port. The actual matching requires finding the LPM or the Longest Prefix Matching [18]. Recently the problem is getting more significance given the anticipated switch from the 32-bit IPv4 to the 128-bit IPv6 [1].

The main research streams that deal with the packet forwarding problem are algorithm-based and hardware-based. The most well-known algorithm-based solutions use the binary trie data structures [6, 16, 18, 20]. A trie is: "a tree-like data structure allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching" [18]. Figure 1(a) shows and example of an 8-bit address IP forwarding table, where $a, b, \cdots, m$ are symbols given to the prefixes for identification. In Figure 1(b) we show the equivalent binary trie of the IP forwarding table given in Figure 1(a). The main advantages of a trie-based solution are that they provide simple time and space bounds. However,



	Prefix	Port
a	000*****	0
b	000101**	1
c	0001111*	2
d	0010****	0
e	00111***	2
f	0110****	0
g	01111***	2
h	1*******	0
i	1001****	0
j	11011***	2
k	110101**	1
l	111101**	1
m	1111111*	2

Fig. 1. (a) An example of an 8-bit address space forwarding table. (b) Its binary trie representation.

with the 128-bit IPv6 prefixes, both trie height and enumeration become an issue when the prefixes are stored inside the nodes.

Hardware-based packet forwarding engines are divided into many classes. The first class uses the Ternary Content Addressable Memory (TCAM), which has been the *de facto* standard for the packet forwarding application [18, 19]. A TCAM is a fully-associative memory that can store 0, 1 and don't care bits. In a single clock cycle, a TCAM chip finds the longest prefix that matches the address of the incoming packet by searching all stored prefixes in parallel. Nevertheless, TCAM has serious deficiencies: high power consumption, poor scalability to long IPv6 prefixes and lower operating frequency compared to other memory technologies [2].

The class of hash-based hardware packet forwarding engines has become popular recently [9–11]. The hash-based engines are promising because they offer constant search time. However, inefficiency rises when two or more keys are mapped to the same bucket, which is called "collision". One way to handle collisions is by *chaining*, which makes each bucket of the the hash table a linked list. The most obvious downside of chaining is the unbounded memory access time [5].

The last class of hardware solutions is based on the multibit trie representation of the forwarding table. In a multibit trie, one or more bits are scanned in a fixed or variable *strides* to direct the branching of the children [18, 22]. Figure 2(a) shows a three-level fixed stride equivalent of the IP Figure 1(a). The trie root in Figure 2(a) has 8 children or rows, since we use the first 3 bits at

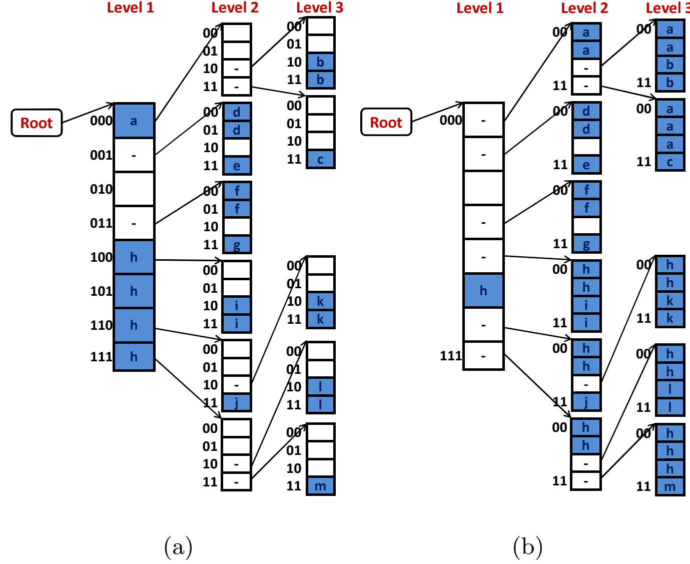


Fig. 2. (a) Multibit trie for Figure 1(a) with strides $\{3, 2, 2\}$. (b) Its leaf-pushed trie.

level one for branching. Each node in the multibit trie is either empty or stores a prefix. Multibit trie-based solutions have the following features: (1) they are easily mapped into a hardware pipeline [2], (2) they reduce the lookup time greatly compared to binary trie [13], and (3) they have low power consumption [13].

A multibit trie reduces the memory lookup time of a unibit trie by decreasing its height [18, 22]. The prefixes in a multibit trie have to be expanded into a set of allowed lengths, through a process called "Controlled Prefix Expansion" (CPE) [20]. Gupta et al. [8] propose a two-level hardware multibit trie, of 24 and 8 bits strides, for IPv4 packet forwarding. The scheme's lookup time is 2 memory cycles at the memory cost of at least 33MB. In general, the strides of a k-level multibit trie will be denoted by $s = \{s_1, \cdots, s_k\}$, where s_l is the number of bits used at level l.

Prefixes can be represented as *address ranges* that in some cases overlap [18]. The prefixes that do not overlap are "disjoint" prefixes [20, 23]. If all prefixes in the forwarding table are disjoint, then we call them "independent" [23]. The independent prefix sets are important because there is only one prefix that matches any incoming packet, thus avoiding the LPM calculation. Any prefix set is transformed into an independent set using a technique called *leaf pushing* [20, 23]. Figure 2(b) shows the leaf-pushed multibit trie for Figure 1(a), where we copy (or push) the prefixes from the intermediate nodes to the leaves. For example, prefix '**a**' in Figure 2(a) is copied two times at level 2 and four times at level 3 in Figure 2(b).

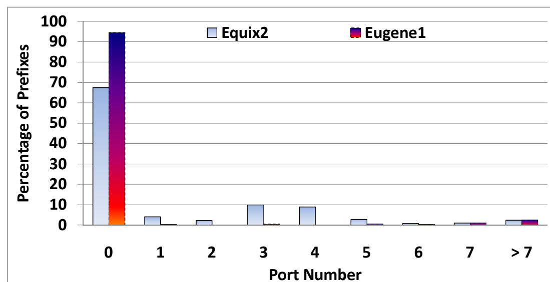


Fig. 3. Prefixes distribution of Eugene1 and Eqix2 vs. output ports numbers.

In this paper we propose a novel IP forwarding scheme based on the compressed multibit trie framework. The main goal is to avoid any prefix matching during the IP lookup process while achieving scalability to the 128-bit IPv6 and relaxing the memory requirement. We reduce the memory footprint by introducing a new two-phase inter-node compression algorithm. Unlike existing compression algorithms [6,16], we do not use any encoding or bitmaps that have adverse effects on the run time and usually complicate the incremental update process. By using an SRAM pipelined architecture, we estimate that our scheme can process 4.9 Giga packets per second and at the same time use less than 1.0 MB of memory for real IPv6 tables with 10.4K prefixes on average.

The rest of this paper is organized as follows: In Section 2 we describe our new IP forwarding scheme.The experimental results and evaluation are given in Section 3. Before we conclude and discuss the future work in Section 5, we talk about prior art in Section 4.

2 Our New Forwarding Scheme

An interesting observation is that the outcome of any IP lookup operation is an interface (or output port) number [4,22]. Real-life backbone routers contain a relatively small number of output ports (usually a few tens). Note that there is a difference between the next hop information, which is an IP address of a router, and the output port which is just a physical port. The routers assign many next hop addresses to one output port [4,22]. This means that the number of unique next hop information is greater than the number of output ports. Throughout this work we use both terms "next hop information" and "output port" interchangeably.

Furthermore, we find that the distribution of the prefixes among the output ports exhibits a certain skewness. Figure 3 plots the distribution of the prefixes against the output ports of tables "Eugene1" and "Equix2", chosen as representatives from the IPv6 tables listed in Table 1 (Section 3). We find that 94% of the prefixes share port 0 as for Eugene1, while 68% prefixes share port 0 for Equix2. Hence, if we replace the prefixes in a trie representation of the forwarding table with their associated port numbers, we have a great opportunity to reduce the

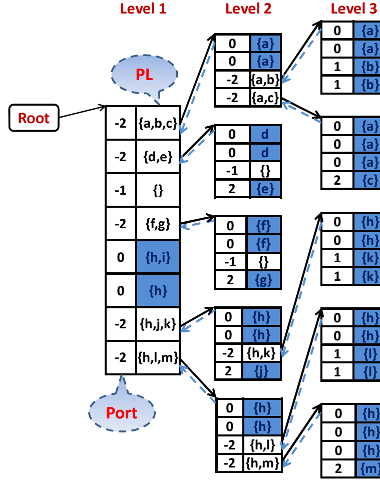


Fig. 4. Uncompressed multibit trie for Figure 1(a).

total number of nodes in this trie using inter-node compression. However, before we describe our inter-node compression algorithm, we build a leaf-pushed, port-based multibit trie which is trimmed to remove redundant leaves.

2.1 Constructing The Uncompressed Multibit Trie

Figure 4 depicts our trie before applying our inter-node compression technique for the forwarding table in Figure 1(a). We call this trie "Trimmed, leaf-pushed, port-based multibit trie", since it is a leaf-pushed multibit trie in which we replace the prefixes with their output port numbers. If there is a subtrie that has the same port number for all of its leaves, then we trim it keeping in its place only one leaf with that port number. Herein, we will refer to this trie as "uncompressed" trie since our final trie is compressed (Section 2.2).

Each node at level l of a k-level multibit trie with strides $s = \{s_1, \cdots, s_k\}$ is a data structure of type "**UTN**", uncompressed trie node, which consists of a single dimension array, $T[\,]$ of size 2^{s_l}, plus a back pointer, *bp*, to the node that points to the current node. For example, the trie in Figure 4 has $k = 3$ levels and a set of strides $s = \{3, 2, 2\}$. Each row in $T[\,]$ contains three variables: PL, a list of prefixes to keep the prefixes that are mapped to this row, *port*, to store the port of this row if all of its PL prefixes have the same port, and *ptr* to store a forward pointer to a next level UTN node (if any). The *port* field is set to -1 if the row is empty and to -2 if the row has a pointer to the next level. Note that the first column in each trie node in Figure 4 is almost identical to the leaf-pushed trie of Figure 2(b) after replacing the prefixes with their ports.

The idea is simple: construct a leaf-pushed multibit trie, then replace each prefix with a port number. During this process, sometimes the prefixes that are mapped to a certain row have identical port numbers. As an example, consider

Algorithm 1 Uncompressed Multibit Trie Construction Algorithm.

```
UTN * BuildNode(Prefixes List SP, int l, UTN *cp)
{
  Define N = new UTN node with N.T[ ] = new array of size 2^{s_l}
  and N.bp = cp /*record back pointer*/
  for (each p ∈ SP) do {
    if(|p| < S_l), then {
      Expand p to length S_l, add the expanded prefixes
      to SP and remove any redundancy } }
  for (each p ∈ SP) do {
    r = row index specified by bits S_{l-1} to S_l of p
    N.T[r].PL = N.T[r].PL ∪ {p} }
  for(r = 1; r ≤ 2^{s_l}; r++){ /*each row*/
    if(N.T[r].PL is empty), then {
      N.T[r].port = -1 } /*empty row*/
    else {
      if(∀(p, q) ∈ N.T[r].PL, p.port == q.port), then {
        N.T[r].port = p.port }
      else {
        Let p_L be the longest prefix in N.T[r].PL
        if(|p_L| ≤ S_l), then{
          N.T[r].port = p_L.port }
        else { N.T[r].port = -2 /*mark as pointer*/
          N.T[r].ptr = BuildNode(N.T[r].PL, l + 1, N.T[r]) }
    } } }
  return N
}
```

row number 4 at level 1 of Figure 4 where both prefixes h and i map to port '0'. Thus, we do not need to expand the trie to the second level as shown in Figure 2(b); instead we just stop at level 1 for this row. Thus, we remove the second level leaf that originally stemmed from row 4 of level 1 and replace it with one port, 0, at this row. Hence, the number of nodes in the leaf-pushed trie in Figure 2(b) is reduced. The dotted arrows in Figure 4 represent back pointers, while the solid ones represent regular next level, *forward*, pointers. The back pointers are essential for our inter-node compression Algorithm 2 (Section 2.2).

Algorithm 1 has one function, **BuildNode()**, which takes three arguments: a list of prefixes SP, an integer l = level number, and a UTN current pointer, *cp*. We start by passing a prefix list SP = all the forwarding table prefixes, $l = 1$ and a $NULL$ pointer from the *main()* function to $BuildNode()$, which returns a pointer, *Root*, to a Uncompressed trie. $BuildNode()$ begins by allocating a new UTN pointer, N, as a new node, records its back pointer and allocates its $T[\,]$ as an array of size 2^{s_l}. It expands any prefix from SP that has a length less than S_l, a cumulative stride[1], and adds only the unique prefixes to the original

[1] A cumulative stride, $S_l = \sum_{i=1}^{l}(s_i)$, is the number of bits that are used till trie level l.

prefix set SP. In other words, if any of the expanded prefixes already exist in SP, we delete the expanded prefix since it inherits its original prefix's length. Each prefix is mapped to a row r, where $r = 1, \cdots, 2^{s_l}$, in the $N.T[\,]$ array and is stored in its associated prefix list, $N.T[r].PL$.

If no prefix is mapped to a certain row, r, we set $N.T[r].port$ to -1 (indicating an empty row). When a packet is matched at this row we apply the default route [4]. If all the prefixes that are stored in r have the same output port number, we simply set $N.T[r].port$ to that port. In case all the prefixes that are mapped to r have lengths less than or equal to S_l, we choose the longest prefix, say p_L, and set $N.T[r].port$ to $p_L.port$. Note that we define $|p|$ to be the length of prefix p and $|SP|$ to be the size of the set of prefixes SP. The recursion in Algorithm 1 occurs when there is at least one prefix that is mapped to r and has a length longer than S_l. In this case, we set $N.T[r].port$ to -2 (indicating that this row has a pointer to the next level) then we set $N.T[r].ptr$ to a new instance of the $BuildNode()$ by passing the set of prefixes that are mapped to this row, $N.T[r].PL$, the next level number, $l+1$, and the current table row pointer, $N.T[r]$.

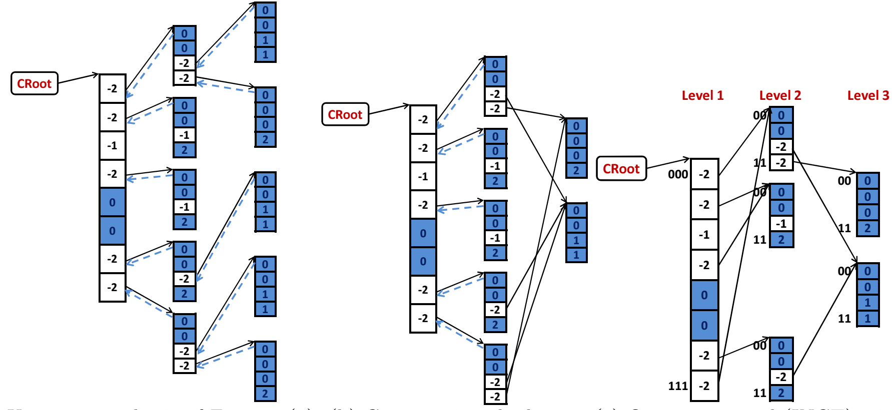
2.2 Inter-Node Compression

In Algorithm 2 we construct a compressed trie from our uncompressed trie generated by Algorithm 1. Our algorithm aims at removing redundancy in trie nodes by checking if there exist two or more nodes at the same level that have identical contents. Since our compression algorithm combines different nodes at the same level, we call it "inter-node" compression and the resulting compressed trie is called *Inter-Node Compressed Trie* (**INCT**).

We start with a copy of our uncompressed trie at level $l = k$ and scan each pair of tables at this level to see if there exists identical tables. If we find two identical nodes, we delete one of them and use its back pointer to update the forward pointer of its parent node. We keep doing this at each level in a backward fashion till we stop at the root. Figure 5(a) represents the three-level trie of Figure 1(a), while Figures 5(b) and (c) depict how we perform compression on this trie level by level. We do not show the back pointers in Figures 5(b) and (c) as they are not needed after the compression. We define the "compression ratio" as the percentage of the total number of trie rows after the compression divided by the total number of trie rows before the compression. The compression ratio in this example is $\frac{28}{48} = 58.3\%$.

The main idea of Algorithm 2 is to make a copy, *CRoot*, from the uncompressed trie, *Root*, then compress it and move the compressed trie without the back pointers and the PL fields to the forwarding plane. For this purpose we need to define $noPtrs[\,]$ as a global array of counters of size k to keep track of how many nodes we have per trie level. In addition, we define an array of UTN pointers, $P[\,][\,]$, that stores the pointers to all the nodes for each trie level, which is used only during the compression and is deleted afterwards.

Function **CompressTrie**() of Algorithm 2 does the actual compression. It starts from the uncompressed trie copy, *CRoot*, then builds the $P[\,][\,]$ arrays.



(a) Uncompressed trie of Figure 1(a). (b) Compressing the leaves. (c) Our compressed (INCT) trie.

Fig. 5. Compressed trie construction from its corresponding uncompressed trie.

Algorithm 2 The Inter-Node Compression Algorithm.

CompressTrie(UTN $*CRoot$)
{
Construct the arrays $noPtrs[\,]$, $P[\,][\,]$
for($l = k$; $l > 1$; $l - -$) {
for($i = 1$; $i < (noPtrs[l] - 1)$; $i + +$) {
for($j = i + 1$; $j < noPtrs[l]$; $j + +$) {
if(**IdenticalNodes**($P[l][i]$, $P[l][j]$)) **then**{
/*use P[l][j]'s back pointer to point to $P[l][i]$*/
$*(P[l][j].bp).ptr = P[l][i]$
Delete node $P[l][j]$ }
} } }
}

The function then sequentially scans the trie levels starting from the leaves to the root to see if there are two identical nodes, which is done by calling **IdenticalNodes()**. $IdenticalNodes()$ is a function that takes two UTN node pointers as arguments and returns $true$ if and only if the two nodes have exactly the same contents, otherwise, it returns $false$. If we find two identical nodes $P[l][i]$ and $P[l][j]$, we use the back pointer of $P[l][j]$ to adjust its forward pointer to point to $P[l][i]$ and delete $P[l][j]$.

2.3 INCT Packet Lookup

In this section we show how to search for a port number in our INCT trie-based IP forwarding scheme. The great advantage is that we do not have to match the incoming packet address against any prefix. During a packet lookup, we split the destination address into k chunks and use each chunk as a row index, r_l,

Table	Size	H	Table	Size	H
Equix1	3,180	9	Linx2	37,282	13
Equix2	3,215	9	Quagga1	3,464	7
Eugene1	3,211	16	Quagga2	3,299	4
Eugene2	3,233	15	Wide1	5,412	2
Linx1	36,366	13	Wide2	5,470	2

Table 1. Statistics of IPv6 tables on June 2010. **H** is the number of output ports.

where $l = 1, \cdots, k$, to a certain table. We start from the $CRoot$, and for each row $r_l \in CRoot.T[\ ]$, if $Root.T[r_l].port \geq 0$, we direct the packet to this port and we stop scanning the trie. However, if $CRoot.T[r_l].port$ equals -1, then we apply the default route. Finally, if $CRoot.T[r_l]\ .port$ equals -2, then we use $CRoot.T[r_l].ptr$ to move to the next next level and recursively repeat the process until either find a port or an empty row.

3 Experimental Evaluation

To evaluate our proposed ideas we built a simulation environment written in C++ and employed 10 real IPv6 files dated June 2010 from the routing information service (RIS) project [17]. Table 1 lists the tables, along with their sizes and their number of unique next hops.

The memory requirement at each row of either uncompressed trie or INCT trie is encoded in 3 bytes, where we use 2 bytes for a pointer and one byte for a port number. An empty row is encrypted by resetting all 3 bytes to zeros. Two bytes per pointer gives us the ability to address $2^{16} = 65.5$K nodes per level.

3.1 INCT Evaluation

Selecting the Strides. A common practice in pipelined trie forwarding engines is to use a large initial stride [12]. This is due to the fact that most prefixes in Internet IPv6 forwarding tables are of length 24 bits or longer [4, 14, 22]. In this work we follow this common practice. We aim to have a small number of trie levels to limit the delay incurred by any incoming packet.

First, we select the number of strides for two representative IPv6 forwarding tables to find the best number of strides. Figure 6 shows the result of varying the number of strides from 5 to 9 for the two files Linx1 and Eugene2. We find that the same trend that having a larger number of strides leads to lower memory utilization. However, since we want to reduce the packet's delay, we choose $k = 7$ as a tradeoff between memory utilization and delay. To select the actual strides' values, we use a simple brute-force heuristic to reduce the total memory requirements of our INCT trie as we mentioned before. In the following subsections we will use the notation "$Xyz(m)$" to distinguish different

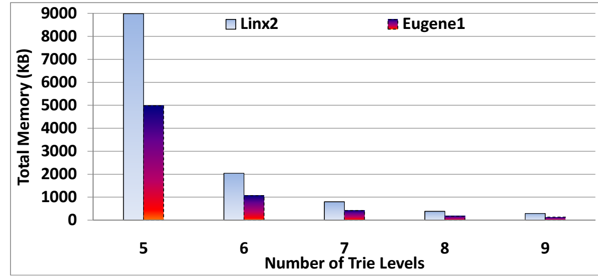


Fig. 6. Number of INCT trie levels vs. total memory (KB) for Linx2 and Eugene1.

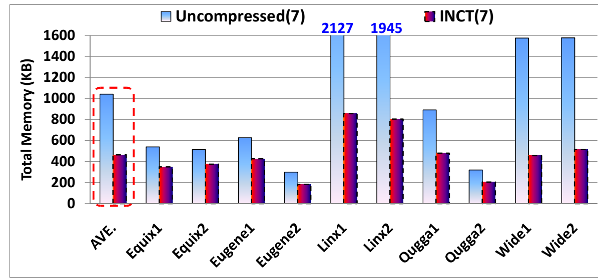


Fig. 7. Memory (KB) of INCT(7) vs. Uncompressed(7) trie for tables given in Table 1.

trie schemes, where Xyz is the trie scheme name and m is the number of trie levels (i.e., height). For example, Uncompressed(7), means our uncompressed trie with a height of 7.

INCT Performance. Figure 7 shows the memory requirements of both INCT(7) and Uncompressed(7) trie for the IPv6 tables given in Table 1. The average total memory for INCT(7) is 466KB, while Uncompressed(7) trie has an average of 1.0MB. This means that we achieve on average a compression ratio of 44.7% or an average space saving of 55.3%. The maximum memory requirement belongs to table Linx1, 857KB for INCT(7) and 2.0MB for Uncompressed(7) trie. The smallest space savings is 27% for table Equix2, while the largest space savings is 71% for table Wide1. The actual compression ratio varies from file to file due to the distribution of the output ports among the prefixes.

INCT vs. Other Compression Schemes for IPv6. Figure 8 shows the memory utilization of MIPS(57), binary INCT(57), Lulea(6) (which is almost like the Tree Bitmap scheme [7]) and INCT(6). Since IPv6 is currently using only 64 bits out of its 128 bits for actual address prefixes while the other 64 bits are to identify MAC address [14, 15], we assumed that Lulea IPv6 version would have 6 strides: $\{16, 16, 8, 8, 8, 8\}$. In addition, we assumed that MIPS's stride set would start by an initial stride of 8 bits then scans the rest of the 64 bits, 56 bits, one bit at a time. Note that INCT(6) is using the same strides as Lulea(6), while binary INCT(57) is using the same strides of MIPS(57).

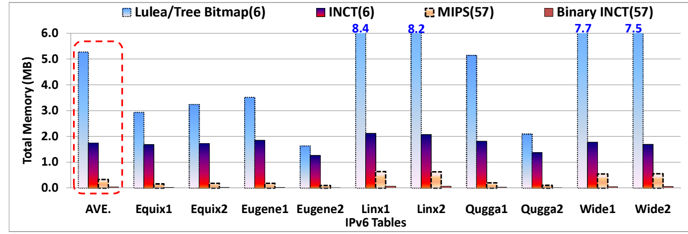


Fig. 8. Memory (MB) of Lulea(6)/Tree Bitmap, INCT(6), MIPS(57) and Binary INCT(57) for tables in Table 1.

MIPS [23] is the first technique that utilized the limited number of ports in Internet routers to lower the number of prefixes in a forwarding table. The main goal of MIPS is to store the prefix set inside a TCAM chip in any order. MIPS builds a leaf-pushed binary trie to obtain an independent set of prefixes, replacing prefixes by their next hops which are then replaced with their corresponding port numbers. As a secondary effect MIPS reduces the number of prefixes needed to be stored. Our trie construction algorithm, Algorithm 1, can be considered as a generalization of the MIPS trie construction algorithm for the multibit trie case. MIPS also keeps its trie in the control plane for update handling, while it retains a copy of the trie prefixes in the TCAM lookup engine. Our scheme does not store the prefixes in the fast forwarding data plane.

The performance is measured as the total memory used by each scheme. In general, MIPS(57)'s average total memory is 339KB, while Lulea(6)'s is 5.3MB. Binary INCT(57) is smaller by 88% than MIPS(57) on average and INCT(6) is smaller by 67% than Lulea(6).

3.2 INCT Forwarding Engine Performance Estimation

In this section we estimate the actual processing rate of our INCT packet forwarding engine using the standard CACTI memory version 5.3 simulator [21]. We use the high performance 32nm SRAM technology with one read port and one write port.

The CACTI results of the pipelined architectures are shown in Figure 9. For each IPv6 table we choose the maximum size of each trie level to be simulated. This is why the total size of each trie is larger than its previously reported average size. The IPv6 INCT(7) saves 60.6% of area and 41.5% of total read dynamic energy over the Uncompressed(7). The stage frequency equals the minimum of the maximum operating frequency among the trie levels, which is 4.9GHz. In other words, the pipelined architecture's estimated throughput is 4.9Giga packets per second, which means a loss of 8% comparing to Uncompressed(7). For minimum packet size of 80 bytes, the INCT(7) throughput is $4.9 \times 80 \times 8 =$ 3.1Tbps (Tera bit per second). The total delay incurred by a packet, which is the summation of the access time of each pipeline stage, is 2.9ns for INCT(7).

	Uncompressed(7)	INCT(7)	Savings and Losses %
Total RAM Size (MB)	2.11	0.85	59.8
Total Access Time (ns)	3.74	2.85	23.8
Pipeline Stage Frequency (GHz)	5.29	4.90	-7.4
Total Read Dynamic Energy (nJ)	0.09	0.05	3.0
Total Read Dynamic Power at Max Freq. (W)	0.54	0.29	45.9
Total Area (mm^2)	5.42	2.14	60.6

Fig. 9. The CACTI simulations for INCT(7) vs. Uncompressed(7).

4 Prior Art and Related Work

Many optimizations are suggested to reduce the binary trie lookup time like the path-compressed trie which benefits from the fact that a binary trie may have long sequences of single-child nodes, then skip scanning some bits to reduce average lookup time [18]. Crescenzi et al. [18] use full prefixes expansion to 32 bits, then construct two-level multibit tries of 16 bits each. Their compression technique is based on repetitions elimination of the LPM of each subtrie at the second level only. They use 1.2MB to store a lookup table of 40K prefixes.

The Lulea algorithm [6] uses *bitmap* compression to eliminate data redundancy in three levels, leaf-pushed tries with fixed strides of $s = \{16, 8, 8\}$ [22]. The scheme has two data structures per node: bitmap and an array that contains the compressed data. For each row of a trie, Lulea stores '1' in the bitmap if the current and previous row values are not equal while pushing the current value to the compressed data vector, otherwise it stores '0'. Though the Lulea scheme is reported to store 32K prefixes in 160KB memory, it has a lookup time between 4 and 12 memory cycles and suffers from very high update time [6, 22]. Eatherton et al. [7] modify the Lulea scheme to facilitate incremental updates while keeping the compression ratio as low as that of Lulea's. The authors propose two bitmaps per trie node, one for all internally stored prefixes and one for external pointers, which allow them to reduce the update time through avoiding leaf pushing [7, 22]. They store a table of 41K prefixes in 450KB.

Note that all these schemes are "intra-node" compression schemes, while our scheme is an "inter-node" compression scheme. Though these preceding algorithms are devised mainly to handle the 32-bit IPv4, they could be utilized also for the 128-bit IPv6 address [14]. In fact, very few trie-based solutions are explicitly proposed to handle IPv6 [3, 15]. In [15], the authors propose a hybrid data structure that combines binary tree (trie), a segment table and a route bucket (hash table) to store an average of 645 real IPv6 prefixes in 390KB with a worst case access latency of six memory cycles. The authors in [3] propose an architecture similar to [15] in the sense that they use a hybrid data structure comprised of: direct lookup table, hash table and multibit trie that uses a new Tree bitmap

compression. They achieve a throughput of 160Gbps utilizing on-chip SRAM and off-chip DRAM memory of 4.9MB for synthetic IPv6 tables.

Recently, many researchers are proposing trie-based advanced pipeline architectures [2, 12, 13]. Their architecture optimizations are based on the minimization of the stage maximum memory, the minimization of the total memory, or the maximization of the throughput. Most of these solutions use dynamic programming algorithms to map trie nodes from to pipeline stages. The work in [13] constructs a pipelined architecture from fixed stride tries since variable stride tries are hard to implement and maintain. The authors provide dynamic programming algorithms to find the optimal multibit trie for a given table. The optimization in [13] is based on minimizing the maximum memory required for a single stage. The pipeline depth is between 2 and 8 stages and each stage uses between 60KB and 80KB for an IPv4 forwarding table of 100K prefixes.

Jiang and Prasanna [12] addressed the same problem with a different strategy. Their architecture consists of multiple pipelines that have between 20 and 25 stages, each with 18KB of RAM. The main idea is to have more stages than the actual number of the trie levels for flexibility. They report an overall a throughput of 3.2Tbps, which is boosted by using an intelligent caching technique that allows processing 8 packets in parallel. The aforementioned schemes are orthogonal to our proposed scheme and can be used to balance memory distribution among stages before applying our compression.

5 Conclusions and Future Work

In this paper we introduced a novel packet forwarding scheme that uses multibit trie inter-node compression to reduce the total memory. We showed that our scheme has a better compression ratio than MIPS, Lulea and Tree bitmap compression schemes. In addition, most previously proposed schemes store their prefixes (or a pointer to them) in the forwarding trie, which we do not. On average, our pipelined INCT architecture, INCT(7), consumes 466KB and can achieve a throughput of 3.1Tbps.

We believe that this work could be applied to other packet processing domains such as packet classification and deep packet inspection. Also, we need to consider systematic methods to select the trie strides rather than just reducing the memory. Since our INCT utilizes an inter-node compression technique, we can apply other intra-node compression techniques (e.g., Lulea [6]) in addition to achieve more compression ratio. Furthermore, we can increase the throughput by using smart caching techniques that take into account the traffic characteristics as well as the trie shape.

Acknowledgment

This work was supported in part by a NSF grant (CCF-0952273).

References

1. T. Arano. IPv4 Address Report. http://www.potaroo.net/tools/ipv4/index.html, 2010. Potaroo Projection.
2. F. Baboescu, D. T., G. Rosu, and S. Singh. A Tree Based Router Search Engine Architecture with Single Port Memories. *ACM Sigarch Com. Arch.*, 33(2), 2005.
3. M. Bando and J. Chao. Flashtrie: Hash-based Prefix-Compressed Trie for IP Route Lookup Beyond 100Gbps. IEEE Infocom, 2010.
4. H. J. Chao and B. Liu. *High Performance Switches and Routers.* Wiley, 2007.
5. T. Cormen, C. Leiserson, R. Rivest, and C. Stien. *Introduction to Algorithms.* McGraw Hill, 2003.
6. M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. ACM Sigcomm, 1997.
7. W. Eatherton, G. Varghese, and Z. Dittia. Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *ACM Sigcomm Comp. Rev.*, 34(2), 2004.
8. P. Gupta, S. Lin, and N. Mckeown. Routing Lookups in Hardware at Memory Access Speeds. IEEE Infocom, 1998.
9. M. Hanna, S. Demetriades, S. Cho, and R. Melhem. CHAP: Enabling Efficient Hardware-based Multiple Hash Schemes for IP Lookup. IFIP Networking, 2009.
10. M. Hanna, S. Demetriades, S. Cho, and R. Melhem. Progressive Hashing for Packet Processing Using Set Associative Memory. IEEE/ACM ANCS, 2009.
11. M. Hanna, S. Demetriades, S. Cho, and R. Melhem. Advanced Hashing Schemes for Packet Forwarding Using Set-Associative Memory Architectures. *Journal of Distributed and Parallel Computing (JPDC), Elsiver*, 71:1–15, 2011.
12. W. Jiang and V. Prasanna. Multi-Terabit IP Lookup Using Parallel Bidirectional Pipelines. ACM Computing Frontiers, 2008.
13. K. S. Kim and S. Sahni. Efficient Construction of Pipelined Multibit-Trie Router-Tables. *IEEE Trans. on Comp.*, 56(1), 2007.
14. Y. K. Li and D. Pao. Comparative Studies of Address Lookup Algorithms for IPv6. IEEE ICACT, 2006.
15. Z. Li, D. Zheng, and Y. Ma. Tree, Segment Table, and Route Bucket: A Multistage Algorithm for IPv6 Routing Table Lookup. IEEE Infocom, 2007.
16. S. Nilsson and G. Karlsson. IP-Address Lookup Using LC-Tries. *IEEE J. on Sel. Areas in Comm.*, 17(6), 1999.
17. RIS. Routing Information Service. http://www.ripe.net/ris/.
18. M. Ruiz-snchez, E. Biersack, and W. Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network*, 15(2), 2001.
19. D. Shah and P. Gupta. Fast Updating Algorithms for TCAMs. *IEEE Micro Mag.*, 21(1), 2001.
20. V. Srinivasan and G. Varghese. Fast Address Lookups Using Controlled Prefix Expansion. *ACM Trans. Comp. Sys.*, 17(1), 1999.
21. S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1: An Integrated Cache Timing, Power, and Area Model. Technical report, HP Labs.
22. G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices.* Morgan Kaufmann, 2005.
23. G. Wang and N.-F. Tzeng. TCAM-Based Forwarding Engine with Minimum Independent Prefix Set (MIPS) for Fast Updating. IEEE ICC, 2006.