Recursively Partitioned Static IP Router-Tables *

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

Abstract

We propose a method–recursive partitioning–to partition a static IP router table so that when each partition is represented using a base structure such as a multibit trie or a hybrid shape shifting trie there is a reduction in both the total memory required for the router table as well as in the total number of memory accesses needed to search the table. The efficacy of recursive partitioning is compared to that of the popular front-end table method to partition IP router tables. Our proposed recursive partitioning method outperformed the front-end method of all our test sets.

Keywords: Packet forwarding, longest-prefix matching, router-table partitioning.

1 Introduction

An IP router table is a collection of rules of the form (P, NH), where P is a prefix and NH is a next hop. The next hop for an incoming prefix is computed by determining the longest prefix in the router table that matches the destination address of the packet; the packet is then routed to the destination specified by the next hop associated with this longest prefix. Router tables generally operate in one of two modes–static (or offline) and dynamic (or online). In the static mode, we employ a forwarding table that supports very high speed lookup. Update requests are handled offline using a background processor. With some periodicity, a new and updated forwarding table is created. In the dynamic mode, lookup and update requests are processed in the order they appear. So, a lookup cannot be done until a preceding update has been done. The focus of this paper is static router-tables. The primary metrics employed to evaluate a data structure for a static table are memory requirement and worst-case number of memory accesses to perform a lookup. In the case of a dynamic table, an additional metric–worst-case number of memory accesses needed for an update–is used.

In this paper, we propose a method to partition a large static router-table into smaller tables that may then be represented using a known good static router-table structure such as a multibit trie (MBT) [15] or a hybrid shape shifting trie (HSST) [6]. The partitioning results in an overall reduction in the number of memory accesses needed for a lookup and a reduction in the total memory required. Section 2 reviews related work on router-table partitioning and Section 3 describes our partitioning method. Experimental results are presented in Section 4.

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

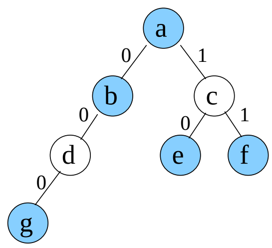
2 Related Work

Ruiz-Sanchez, Biersack, and Dabbous [11] review data structures for static router-tables and Sahni, Kim, and Lu [13] review data structures for both static and dynamic router-tables. Many of the data structures developed for the representation of a router table are based on the fundamental *binary trie* structure [3]. A binary trie is a binary tree in which each node has a data field and two children fields. Branching is done based on the bits in the search key. A left child branch is followed at a node at level i (the root is at level 0) if the ith bit of the search key (the leftmost bit of the search key is bit 0) is 0; otherwise a right child branch is followed. Level i nodes store prefixes whose length is i in their data fields. The node in which a prefix is to be stored is determined by doing a search using that prefix as key. Let N be a node in a binary trie. Let $Q(N)$ be the bit string defined by the path from the root to N. $Q(N)$ is the prefix that corresponds to N. $Q(N)$ (or more precisely, the next hop corresponding to $Q(N)$) is stored in $N.data$ in case $Q(N)$ is one of the prefixes in the router table.

Figure 1 (a) shows a set of 5 prefixes. The $*$ shown at the right end of each prefix is used neither for the branching described above nor in the length computation. So, the length of $P2$ is 1. Figure 1 (b) shows the binary trie corresponding to this set of prefixes. Shaded nodes correspond to prefixes in the rule table and each contains the next hop for the associated prefix. The binary trie of Figure 1 (b) differs from the 1-bit trie used in [15], [13], and others in that a 1-bit trie stores up to 2 prefixes in a node (a prefix of length l is stored in a node at level $l-1$) whereas each node of a binary trie stores at most 1 prefix. Because of this difference in prefix storage strategy, a binary trie may have up to 33 (129) levels when storing IPv4 (IPv6) prefixes while the number of levels in a 1-bit trie is at most 32 (128).

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

(a) 5 prefixes



(b) Corresponding binary trie

Figure 1: Prefixes and corresponding binary trie

For any destination address d, we may find the longest matching prefix by following a path beginning at the trie root and dictated by d. The last prefix encountered on this path is the longest prefix that matches d. While this search algorithm is simple, it results in as many cache misses as the number of levels in the trie. Even for IPv4, this number, which is at most 33, is too large for us to forward packets at line speed. Several strategies–e.g., LC trie [9], Lulea [1], tree bitmap [2], multibit tries [15], shape shifting tries [14], hybrid shape shifting tries [6]–have been proposed to improve the lookup performance of binary tries. All of these strategies collapse several levels of each subtree of a binary trie into a single node, which we call a *supernode*, that can be searched with a number of memory accesses that is less than the number of levels collapsed into the supernode. For example, we can access the correct child pointer (as well as its associated prefix/next hop) in a multibit trie with a single memory access independent of the size of the multibit node. Lunteren [7, 8] has devised a perfect-hash-function scheme for the compact representation of the supernodes of a multibit trie.

Lampson et al.[4] propose a partitioning scheme for static router-tables. This scheme employs a front-end array, *partition*, to partition the prefixes in a router table based on their first s, bits. Prefixes that are longer than s bits and whose first s bits correspond to the number i, $0 \leq i < 2^s$ are stored in a bucket $partition[i].bucket$ using any data structure (e.g., multibit trie) suitable for a router-table. Further, $partition[i].lmp$, which is the longest matching-prefix in the database for the binary representation of i (note that the length of $partition[i].lmp$ is at most s) is precomputed from the given prefix set. For any destination address d, $lmp(d)$, is determined as follows:

1. Let i be the integer whose binary representation equals the first s bits of d. Let V equal $NULL$ if no prefix in $partition[i].bucket$ matches d; otherwise, let V be the longest prefix in $partition[i].bucket$ that matches d.

2. If V is $NULL$, $lmp(d) = partition[i].lmp$. Otherwise, $lmp(d) = V$.

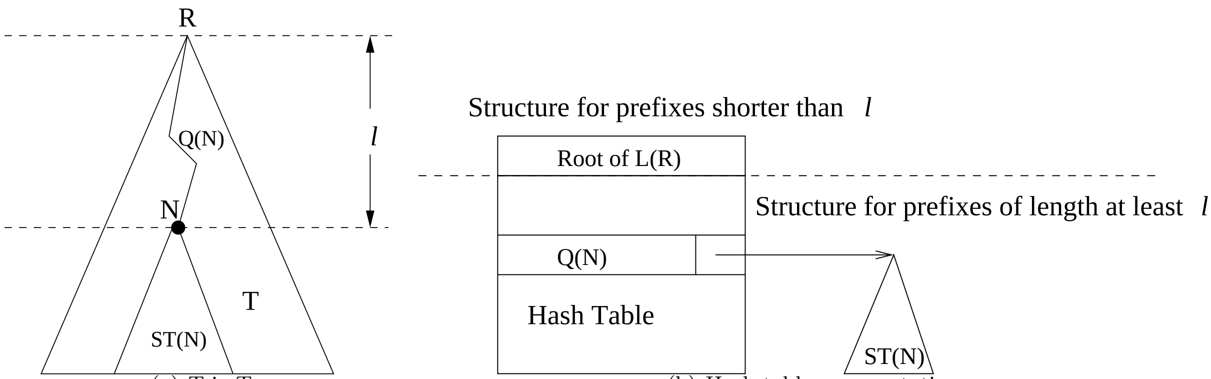
Note that the case $s = 0$ results in a single bucket and, effectively, no partitioning. As s is increased, the average number of prefixes per bucket as well as the maximum number in any bucket decreases. Although the worst-case time to find $lmp(d)$ decreases as we increase s, the storage needed for the array $partition[]$ increases with s and quickly becomes impractical. Lampson et al. [4] recommend using $s = 16$. This recommendation results in $2^s =$ 65,536 buckets. For practical router-table databases that may have up to a few hundred thousand rules, $s = 16$ results in buckets that have at most a few hundred prefixes. Hence, in practice, the worst-case memory accesses to find $lmp(d)$ is considerably improved over the case $s = 0$. However, when $s = 16$, the memory required by the front-end array (exclusive of that required for the base structures that represent each bucket), *partition*, may exceed that required by the base structure when applied to the unpartitioned rule table.

Lu, Kim and Sahni [5] have proposed partitioning schemes for dynamic router-tables. While these schemes are designed to keep the number of memory accesses required for an update at an acceptable level, they may increase the worst-case number of memory accesses required for a lookup and also increase the total memory required to store the structure. Of the schemes proposed by Lu, Kim and Sahni [5], the two-level dynamic partitioning scheme (TLDP) works best for average-case performance. TLDP, like the scheme of Lampson et al. [4], employs a front-end array *partition* with $partition[i].bucket$, $0 \leq i < 2^s$ storing all prefixes whose length is $\geq s$ and whose first s bits correspond to i. Unlike the scheme of of Lampson et al. [4], however, prefixes whose length is less than s are stored in an auxiliary structure X and there is no precomputation of a quantity such as $partition[i].lmp$. The prefixes in X are themselves partitioned using $t < s$ bits and an array $p[i].bucket$, $0 \leq i < 2^t$ stores prefixes whose length is $\geq t$ and $< s$; an auxiliary structure Y is used for prefixes whose length is $< t$. Prefixes in the buckets $partition[i].bucket$ and $p[i].bucket$ as well as those in the auxiliary structure Y are stored using a base structure such as multibit tries. Although, in theory, buckets could be partitioned further, Lu, Kim and Sahni [5] assert that bucket sizes, for their test databases, were sufficiently small that further partitioning resulted in no (or little) performance gain. A drawback of the TLDP scheme is that the partitioning may cause the worst-case number of memory accesses for a lookup to increase. This is because a lookup may require us to search $partition[i].bucket$, $p[j].bucket$ and Y. To overcome this problem, Lu, Kim and Sahni [5] propose precomputing $partition[i].lmp$ only for those i for which $partition[i].bucket$ is not empty. This, however, has an adverse effect on the worst-case performance of an update. Experimental results presented in [5] show that the TLDP scheme leads to reduced average search and update times as well as to a reduction in memory requirement over the case when the tested base schemes are used with no partitioning.

3 Recursive Partitioning

3.1 Basic Strategy

In *recursive partitioning*, we start with the binary trie T (Figure 2(a)) for our prefix set and select a *stride*, s, to partition the binary trie into subtries. Let $D_l(R)$ be the level l (the root is at level 0) descendents of the root R of T. Note that $D_0(R)$ is just R and $D_1(R)$ is the children of R. When the trie is partitioned with stride s, each subtrie, $ST(N)$, rooted at a node $N \in D_s(R)$ defines a partition of the router table. Note that $0 < s \leq T.height + 1$, where $T.height$ is the height (i.e., maximum level at which there is a descendent of R) of T. When $s = T.height + 1$, $D_s(R) = \emptyset$. In addition to the partitions defined by $D_s(R)$, there is a partition $L(R)$, called the *auxiliary* partition, defined by prefixes whose length is $< s$. The prefixes in $L(R)$ are precisely those stored in $D_i(R)$, $0 \leq i < s$. So, the total number of partitions is $|D_s(R)| + 1$. These partitions are called the *first-level* partitions of T.



(a) Trie T (b) Hash table representation

Figure 2: Stride s partitioning of a binary trie T

To keep track of the first-level partitions of T, we use a hash table with a perfect hashing function for the partitions defined by $N \in D_s(R)$. The root of the data structure used for $L(R)$ is placed adjacent, in memory, to this hash table (Figure 2 (b)). The bit strings $Q(N)$, $N \in D_s(R)$ define the keys used to index into the hash table. Although any perfect hash function for this set of keys may be used, we use the perfect hash function defined by Lunteren [7, 8].

We note that when $s = T.height + 1$, the hash table is empty and $L(R) = T$. In this case, T is simply represented by a base structure such as MBT or HSST. When $s < T.height + 1$, the described partitioning scheme may be applied recursively to each of the $|D_s(R)| + 1$ partitions to obtain lower-level partitions. An exception is the case when $N \in D_s(R)$ is a leaf. In this case, the next hop associated with the corresponding prefix is stored directly in the hash table. Each entry in the hash table can, therefore, represent one of four types of information:

Type 1: A partition that is further partitioned into lower-level partitions.

Type 001: A leaf partition.

Type 010: A partition that is represented by a base structure such as an MBT or an HSST.

Type 000: An unused hash table entry.

For type 1 entries, we use 1 bit to identify the entry type. In addition, we store the path $Q(N)$ from the root R to the root N of the partition, the stride for the next-level partition, a mask that characterizes the next-level perfect hash function, and a pointer to the hash table for the next-level partition. Figure 3 shows the schematic for a type 1 entry. For the remaining 3 types, we use three bits to identify the entry type. For entry type 001, we store also $Q(N)$ and the next hop associated with the prefix stored in node N and for type 010, we store $Q(N)$ and a pointer to the base structure used for the partition. Type 000 entries store no additional information.

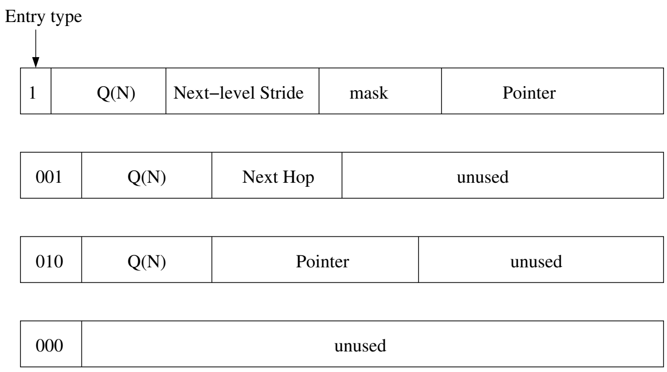


Figure 3: Hash table entry types

Notice that all prefixes in the same first-level partition agree on their first l bits. So, we strip these bits from these prefixes before developing lower-level partitions. In particular, a prefix of length l gets replaced by a prefix of length 0.

Figure 4 gives the algorithm to do a lookup in a router table that has been partitioned using the basic strategy. The algorithm assumes that at least one level of partitioning has been done. The initial invocation specifies, for the first-level partitioning, the stride s, address of first hash table entry, ht, and perfect hash function h (specified by its mask).

```
Algorithm lookup(s, ht, h, d){
   // return the next hop for the destination d
   q = first s bits of d;
   u = remaining bits of d;
   t = ht[h(q)];   // home bucket

   if (t.type == 000 || t.key != q)
      // search auxiliary partition lr(ht) of ht
      return lr(ht).lookup(d);

   // search in bucket t
   switch (t.type) {
   1: // examine next-level partition
      nh = lookup(t.stride, t.pointer, t.mask, u);
      if (nh == NULL) return lr(ht).lookup(d);
      else return nh;

   001: // examine a leaf
        return t.nextHop;

   010: // examine a base structure
        nh = t.pointer.lookup(u);
        if (nh == NULL) return lr(ht).lookup(d);
        else return nh;
   }
}
```

Figure 4: Searching with basic strategy

3.2 Incorporating Leaf Pushing

The worst-case number of memory accesses required for a lookup may be reduced using *controlled leaf pushing*, which is quite similar to the standard leaf pushing used in non-partitioned router tables [15]. In controlled leaf pushing, every base structure that does not have a (stripped) prefix of length 0 is given a length 0 prefix whose next hop is the same as that of the longest prefix that matches the bits stripped from all prefixes in that partition. So, for example, suppose we have a base structure whose stripped prefixes are 00, 01, 101 and 110. All 4 of these prefixes have had the same number of bits (say 3) stripped from their left end. The stripped 3 bits are the same for all 4 prefixes. Suppose that the stripped bits are 010. Since the partition does not have a length 0 prefix, it inherits a length 0 prefix whose next hop corresponds to the longest of *, 0, 01 and 010 that is in the original set of prefixes. Assuming that the original prefix set contains the default prefix, the stated inheritance ensures that every search in a partition finds a matching prefix and hence a next hop. So, the lookup algorithm takes the form given in Figure 5.

```
Algorithm lookupA(s, ht, h, d){
   // return the next hop for the destination d
   q = first s bits of d;
   u = remaining bits of d;
   t = ht[h(q)];   // home bucket

   if (t.type == 000 || t.key != q)
      // search auxiliary partition lr(ht) of ht
      return lr(ht).lookup(d);

   // search in bucket t
   switch (t.type) {
   1: // examine next-level partition
      return lookupA(t.stride, t.pointer, t.mask, u);

   001: // examine a leaf
        return t.nextHop;

   010: // examine a base structure
        return t.pointer.lookup(u);
   }
}
```

Figure 5: Searching with leaf pushing version A

3.3 Optimization

To use recursive partitioning effectively, we must select an appropriate stride for each partitioning that is done. For this selection, we set up a dynamic programming recurrence. Let $B(N, l, r)$ be the minimum memory required to represent levels 0 through l of the subtree of T rooted at N by a base structure such as MBT or HSST; a lookup in this base structure must take no more than r memory accesses. Let $H(N, l)$ be the memory required for a stride l hash table for the paths from node N of T to nodes in $D_l(N)$ and let $C(N, l, r)$ be the minimum memory required by a recursively partitioned representation of the subtrie defined by levels 0 through l of $ST(N)$. From the definition of recursive partitioning, the choices for l in $C(N, l, r)$ are 1 through $N.height + 1$. When $l = N.height + 1$, $ST(N)$ is represented by the base structure. So, from the definition of recursive partitioning, it follows that

$$
\begin{aligned}
C(N, N.height, r) &= \min\{B(N, N.height, r), \\
&\quad \min_{0<l\leq N.height}\{H(N,l) + C(N, l-1, r-1) + \sum_{Q\in D_l(N)} C(Q, Q.height, r-1)\}\},\ r > 0 \quad (1) \\
C(N, l, 0) &= \infty \quad (2)
\end{aligned}
$$

The above recurrence assumes that no memory access is needed to determine whether the entire router table has been stored as a base structure. Further, in case the router table has been partitioned then no memory access is needed to determine the stride and mask for the first-level partition as well as the structure of the auxiliary partition. This, of course, is possible if we store this information in memory registers. However, as the search progresses through the partition hierarchy, this information has to be extracted from each hash table. So, each Type 1 hash-table entry must either store this information or we must change the recurrence to account for the additional memory access required at each level of the partition to get this information. In the former case, the size of each hash-table entry is increased. In the latter case, the recurrence becomes

$$\begin{aligned} C(N, N.height, r) &= \min\{B(N, N.height, r), \\ &\quad \min_{0<l\leq N.height}\{H(N,l) + C(N, l-1, r-2) + \sum_{Q\in D_l(N)} C(Q, Q.height, r-1)\}\},\ r > 0 \quad (3) \\ C(N, l, r) &= \infty,\ r \leq 0 \quad (4) \end{aligned}$$

Recurrences for B may be obtained from Sahni and Kim [12] for fixed- and variable-stride MBTs and Lu and Sahni [6] for HSSTs.

Our experiments with real-world router tables indicates that when auxiliary partitions are restricted to be represented by base structures, the memory requirement is reduced. With this restriction, the dynamic programming recurrence becomes

$$\begin{aligned} C(N, N.height, r) &= \min\{B(N, N.height, r), \\ &\quad \min_{0<l\leq N.height}\{H(N,l) + B(N, l-1, r-1) + \sum_{Q\in D_l(N)} C(Q, Q.height, r-1)\}\} \quad (5) \\ C(N, l, 0) &= \infty \quad (6) \end{aligned}$$

Now, the second parameter l of $C(N, l, r)$ always is $N.height$ and so this second parameter may be dropped. Further optimization is possible by permitting the method used to keep track of partitions to be either a hash table plus an auxiliary structure for prefixes whose length is less than the stride or a simple array with 2^l entries when the partition stride is l (this latter strategy is identical to that used by Lampson et al. [4] for their front-end table). Including this added flexibility, but retaining the restriction that auxiliary partitions are represented as base structures, the dynamic programming recurrence becomes

$$\begin{aligned} C(N, N.height, r) &= \min\{B(N, N.height, r), \\ &\quad \min_{0<l\leq N.height}\{H(N,l) + B(N, l-1, r-1) + \sum_{Q\in D_l(N)} C(Q, Q.height, r-1)\}, \\ &\quad \min_{0<l\leq N.height}\{2^l c + \sum_{Q\in D_l(N)} C(Q, Q.height, r-1)\}\} \quad (7) \\ C(N, l, 0) &= \infty \quad (8) \end{aligned}$$

where c is the memory required by each position of the front-end array. Again, the second parameter in C may be dropped.

Notice that the inclusion of front-end arrays as a mechanism to keep track of partitions requires as to add a fifth entry type (011) for hash table entries. This fifth type, which indicates a partition represented using a front-end array, includes a field for the key $Q(N)$, another field for the stride of the next-level partition, and a pointer to the next-level front-end array. Note also that while our discussion may have given the impression that all base structures in a recursively partitioned router table must be of the same type (i.e., all are MBTs or all are HSSTs), it is possible to solve the dynamic programming recurrences allowing a mix of basic structures.

3.4 Comparison With Other Partitioning Methods

In the following, we highlight the similarities and differences between the recursive partitioning scheme proposed in this paper and the front-end array, prefix partitioning and interval partitioning schemes.

1. Earlier partitioning schemes were limited to either one (e.g., front-end array [4], one-level prefix partitioning [5], interval partitioning [5]) or two (two-level prefix partitioning) levels of partitioning. The strides of the partitioning tables are fixed (e.g., the front-end array of [4] uses a stride of 16) and not data dependent. Recursive partitioning permits multilevel partitioning with strides determined by a dynamic programming recurrence to optimize memory utilization.

2. Earlier partitioning schemes represented all partitions using the same base structure. With recursive partitioning a heterogeneous collection of base structures may be selected to optimize memory utilization.

3. Recursive partitioning permits the use of different methods (front-end array and hash table with auxiliary partition) to keep track of the partitions of a prefix set. Through the use of dynamic programming, one may select the optimal method for each partitioning.

4. When the method to keep track of partitions in limited to front-end arrays and the base structure is a multibit node, recursive partitioning reduces to variable-stride tries [12, 15].

3.5 Implementation Considerations

For benchmarking purposes we assumed that the router table will reside on a QDRII SRAM (dual burst), which supports the retrieval of 72 bits of data with a single memory access. We considered two hash-table designs–36 bit and 72 bit.

In the 36-bit design, we allocated 36 bits to each hash entry. For IPv4, we used 8 bits for $Q(N)$, 2 bits for the stride of the next-level partition, 8 bits for the mask, and 17 bits for the pointer. Although 8 bits were allocated to $Q(N)$, the strides were limited to be between 5 and 8 (inclusive). Hence, 2 bits are sufficient to represent the next-level stride. The use of a 17-bit pointer enables us to index up to 9Mbits ($2^{17} * 72$) of SRAM. For IPv6, the corresponding bit allocations are 7, 2, 7, and 19, respectively. For IPv6, the strides were limited to be between 4 and 7 (hence 7 bits suffice for $Q(N)$ and 2 bits suffice for the next-level stride). The 19-bit pointers are able to index a 36Mbit SRAM. For the next-hop field, we allocated 12 bits for both IPv4 and IPv6.

For the base structure, we used the *enhanced base with end-node optimization* (EBO) version of HSSTs [6] as these were shown to be the most efficient router-table structure for static router tables [6]. Non-leaf EBO nodes have child pointers and some EBO leaf nodes have pointers to next-hop arrays. For child pointers we allocated 10 bits. This allows us to index 1024 nodes. We modified the dynamic programming equations developed in [6] for the construction of optimal EBOs so that EBOs that require more than 1024 nodes are rejected. For next-hop array pointers, we allocated 22 bits. Since, the number of next-hop array pointers is bounded by the number of prefixes in the router table and next-hop arrays are stored in a different part of memory from where we store the rest of the EBO data structure, an allocation of 22 bits for next-hop array pointers suffices for $2^{22} > 4$ million prefixes. For the next hops themselves, we allocated 12 bits.

In the 72-bit design, we allocated 72 bits for each hash-table entry. For both IPv4 and IPv6, we used 17 bits for $Q(N)$, 5 bits for the stride of the next-level partition, 17 bits for the mask, and 19 bits for the pointer; the strides were limited to be between 1 and 17. Also, the next hop for the stripped prefix * (if any) in $L(R)$ is stored in each hash-table entry. A novel feature of the 72-bit design is that the partitioning was enabled so that at each node N, a selection was made between using an $L(R)$ partition represented as an EBO and a (perfect) hash table for the remaining partitions (as described earlier in this paper) and performing a prefix expansion of the stripped prefixes in $L(R) - \{*\}$, distributing these expanded prefixes into the remaining partitions (creating new partitions if necessary), and then constructing a (perfect) hash table for the modified partition set. Type 1 nodes use a designated bit to distinguish between the different hash-table types they may point to. The remaining implementation details are the same as for the 36-bit design.

Although we have described implementations for a specific SRAM, implementations for other SRAMs are obtained easily.

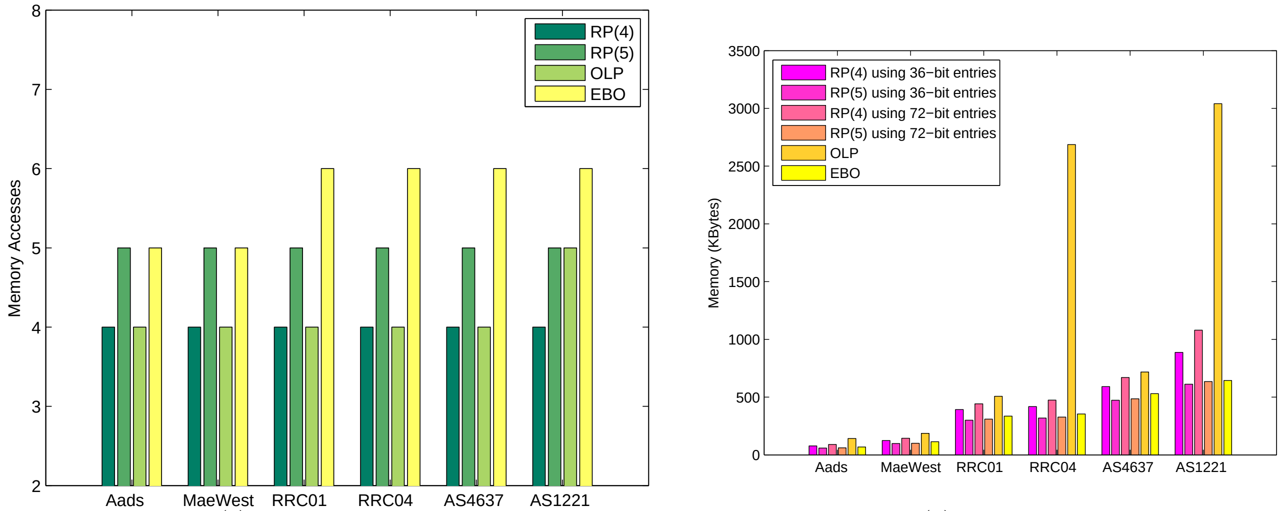
4 Experimental Results

C++ codes for the implementations described in Section 3.5 were compiled using the Microsoft Visual C++ compiler with optimization level `O2` and run on a 3.06 GHz Pentium 4 PC. Our recursive partitioning scheme was compared against a one-level partitioning scheme, OLP, which is a generalization of the front-end array of Lampson et al. [4] and a non-partitioned EBO. OLP does only one level of partitioning (as does [4]) and uses EBO as the base structure. However, unlike [4], which fixes the size of the front-end array to 2^{16}, OLP selects an optimal, data-dependent, size for the front-end array. Specifically, OLP tries out front-end arrays of size 0 and 2^i, $1 \leq i \leq 24$ and determines those sizes that minimize the worst-case number of memory accesses for a lookup; from these sizes, the size that minimizes total memory is selected. Note that using a front-end array of size 0 is equivalent to using no front-end array. We found OLP to be superior, on our data sets, to simply limiting our recursive partitioning scheme so as to partition only at the root level. We did not compare with the partitioning schemes of [5], because these schemes improve average performance at the expense of worst-case performance and our focus in this paper is worst-case performance. The schemes of [5] result in increased memory requirement and worst-case number of memory accesses for a search relative to the base structure.

All of our programs were written so as to construct lookup structures that (a) minimize the worst-case number of memory accesses needed for a lookup and (b) minimize the total memory needed to store the constructed data structure. As a result, our experiments measured only these two quantities. Further, all test algorithms were run so as to generate a lookup structure that minimizes the worst-case number of memory accesses needed for a lookup; the size (i.e., memory required) of the constructed lookup structure was minimized subject to this former constraint.

4.1 IPv4 Router Tables

For test data, we used the six IPv4 router tables Aads, MaeWest, RRC01, RRC04, AS4637 and AS1221 that were obtained from [20, 10, 19]. The number of prefixes in these router tables is 17486, 29608, 103555, 109600, 173501 and 215487, respectively.



(a) number of accesses

(b) total memory

Figure 6: Memory accesses and total memory (KBytes) required for IPv4 tables

Figures 6 plots the number of memory accesses and memory requirement for the tested lookup structures. $RP(k)$ ($K = 4, 5$) denotes the space-optimal recursively partitioned structure that requires at most k memory accesses per search. As can be seen, on the memory access count, $RP(4)$ is superior to EBO on all 6 data sets by 1 or 2 accesses. OLP is superior to EBO by 1 access on 3 of our data sets and by 2 accesses on the remaining 3 data sets; $RP(5)$ is superior to EBO by 1 access on 4 of the 6 data sets. OLP required one more access than $RP(4)$ on the largest data set (AS1221) and tied with $RP(4)$ on the remaining 5.

On all of our test sets, the 36-bit implementation required less memory than required by the corresponding 72-bit implementation. In fact, the 36-bit implementation required between 80% and 98% of the memory required by the 72-bit implementation, the average is 92% and the standard deviation is 6%.

Algorithm	Min	Max	Mean	Standard Deviation
RP(5) using 36-bit entries	0.71	0.80	0.77	0.03
RP(4) using 72-bit entries	1.13	1.25	1.16	0.05
RP(5) using 72-bit entries	0.74	0.82	0.79	0.03
OLP	1.21	6.44	2.64	2.05
EBO	0.75	0.91	0.86	0.06

Table 1: Statistics for IPv4 memory requirement normalized by that for RP(4) using 36-bit entries

Table 1 gives the memory requirement of the lookup structures normalized by that for $RP(4)$ using 36-bit entries. Compared to $RP(4)$ with 36-bit entries, OLP required from 21% to 544% more memory, while EBO required between 9% and 25% less memory. Among all six representations, $RP(5)$ using 36-bit entries was the most memory efficient. Compared to EBO, this implementation of $RP(5)$, used between 5% and 13% less memory; the average reduction is memory required was 10% and the standard deviation as 3%.

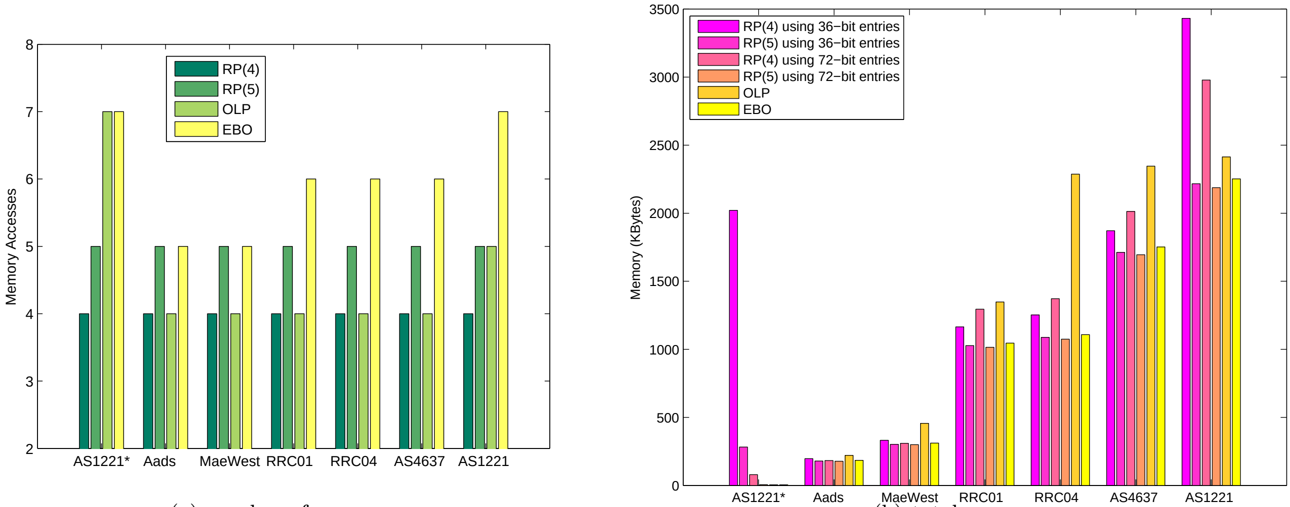
In summary, the 36-bit implementation of $RP(4)$ is superior to OLP on both our metrics–worst-case memory accesses and total memory requirement. It resulted in a 25% to 50% reduction in worst-case memory accesses over EBO. This reduction came at the expense of an increase in required memory between 10% and 37%. The 36-bit implementation of $RP(5)$ improved the lookup time by up to 20% relative to the base EBO structure and reduced total memory by 10% on average. This is quite surprising as the EBO structure is a highly optimized structure.

4.2 IPv6 Router Tables

For our IPv6 experiments, we used the 833-prefix AS1221-Telstra router table that we obtained from [20] as well as 6 synthetic IPv6 tables. Prefixes longer than 64 were removed from the AS1221-Telstra table as current IPv6 address allocation schemes use at most 64 bits [18]. For the synthetic tables, we used the strategy proposed in [17] to generate IPv6 tables from IPv4 tables. In this strategy, to each IPv4 prefix we prepend a 16-bit string comprised of 001 followed by 13 random bits. If this prepending doesn't at least double the prefix length, we append a sufficient number of random bits so that the length of the prefix is doubled. Following this prepending and possible appending, we drop the last bit from one-fourth of the prefixes so as to maintain the 3:1 ratio of even length prefixes to odd length observed in real router tables. Each synthetic table is given the same name as the IPv4 table from which it was synthesized. The AS1221-Telstra IPv6 table is named AS1221* to distinguish it from the IPv6 table synthesized from the IPv4 AS1221 table.

Figures 7 plots the number of memory accesses and memory requirement for our IPv6 data sets. As was the case for our IPv4 experiments, $RP(4)$ was the best in terms of lookup complexity. Particularly, $RP(4)$ required 1 to 3 fewer memory accesses than required by EBO for a lookup. $RP(4)$ and OLP tied on 5 of the 7 data sets; on 1, $RP(4)$ required 3 fewer memory accesses and on the other, it required 1 less access. $RP(5)$ outperformed EBO by 1 or 2 accesses on 5 data sets and tied on the remaining 2.

In contrast to the experiments with IPv4 tables, the 72-bit implementation of recursive partitioning generally required less memory than did the 36-bit implementation. On 11 of our 14 tests ($RP(4)$ and $RP(5)$) with recursive



(a) number of accesses (b) total memory

Figure 7: Memory accesses and total memory (KBytes) required for IPv4 tables

partitioning, the memory required by the 72-bit implementation was less than that required by the 36-bit implementation; it was more on the remaining 3 tests. The memory of recursively partitioned structure using 36-bit hash entries normalized by that required using 72-bit entries ranged from 0.9 to 49.9. We see that the data set AS1221* incurred the largest difference. When AS1221* is excluded, the normalized number for the remaining 6 data sets is between 0.90 to 1.15 (the mean and standard deviation were 1.00 and 0.00).

Algorithm	Min	Max	Mean	Standard Deviation
RP(4) using 36-bit entries	0.90	1.15	1.00	0.11
RP(5) using 36-bit entries	0.74	0.98	0.86	0.10
RP(5) using 72-bit entries	0.73	0.97	0.85	0.10
OLP	0.81	1.67	1.23	0.31
EBO	0.76	1.00	0.87	0.11

Table 2: IPv6 data normalized by the memory required by RP(4) using 72-bit entries. The data set AS1221* is excluded here.

For the data set AS1221*, the 72-bit implementation of $RP(4)$ reduced the memory accesses of EBO by 3 but required 17 times as much memory. The same implementation of $RP(5)$ required 24% more memory than required by the base EBO structure. On the other hand, $RP(6)$ required 3.8 KBytes; a 17% memory reduction accompanied by a reduction in memory accesses of 1. For this data set, OLP yielded no improvement over EBO; that is, OLP wound up using a front-end table of size 0. For the remaining 6 data sets, $RP(5)$ required slightly less memory than EBO. On 5 of the 6 data sets, OLP required more memory than did $RP(4)$. On the sixth data set, AS1221, OLP took less memory. However, when the same budget for worst-case memory accesses was used, $RP(5)$ using 72-bit entries required 9% less memory than OLP on AS1221. Table 2 presents the statistics normalized by the memory required by $RP(4)$ using 72-bit entries for the remaining 6 data sets. As can be seen, the memory of EBO normalized by $RP(4)$ using 72-bit entries ranged from 0.76 to 1.00, with the mean and standard deviation being 0.87 and 0.11. The corresponding normalized numbers for OLP were 0.81, 1.67, 1.23, and 0.31.

5 Conclusion

Recursive partitioning is superior to the front-end table scheme [4] commonly used in conjunction with base router-table data structures. Although we did not do a direct comparison with the standard 16-bit front-end table scheme, we did compare with its generalization OLP, which uses a front-end table that minimizes total memory subject to minimizing the worst-case number of memory accesses per lookup. By design, OLP cannot be inferior to the employed base structure (in our case EBO). OLP improved the lookup performance of EBO by 1 or 2 memory accesses on all but one of our test sets. In all cases, the improvement in lookup performance came at the expense of increased memory reaquirement; for the RRC04 IPv4 data set, OLP reduced the memory accesses per lookup by 2 but required 6.7 times the memory. $RP(4)$ improved the lookup performance of EBO by 1 to 3 memory accesses on all our data sets. On all test sets where $RP(4)$ and OLP resulted in the same lookup performance, $RP(4)$ took less memory than did OLP. For example, on the RRC04 IPv4 data set, the 36-bit implementation of $RP(4)$ took 15.5% of the memory taken by OLP; it took only 18% more memory than EBO while reducing the worst-case memory accesses from 6 to 4.

While both OLP and recursive partitioning are able to improve the lookup performance of EBO, OLP does this with a much larger memory cost. Our experiments demonstrate the superiority of recursive partitioning over even a generalized version of the standard front-end array method. For IPv4 tables, recursive partitioning with 36-bit entries is superior to using larger hash-table entries (e.g., 72 bits) while for IPv6 tables, 72-bit entries often resulted in reduced memory requirement. Although, not reported in Section 4, using even larger hash-table entries (e.g., 144 bits) resulted in no reduction in memory required by either $RP(4)$ or $RP(5)$ for our IPv4 and IPv6 test data. Further, we expect the results reported in this paper to carry over to the case when base structures other than EBO (e.g., multibit tries) are employed.

References

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

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

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

[4] B. Lampson, V. Srinivasan, and G. Varghese, IP lookup using multi-way and multicolumn search, *IEEE INFOCOM*, 1998.

[5] H. Lu, K. Kim, and S. Sahni, Prefix- and interval-partitioned dynamic router tables, *IEEE Trans. on Computers*, 54, 5, 2005, 545-557.

[6] W. Lu and S. Sahni, Succinct representation of static packet classifiers, University of Florida, 2006.

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

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

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

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

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

[12] S. Sahni and K. Kim, Efficient construction of multibit tries for IP lookup, *IEEE/ACM Trans. on Networking*, 11, 4, 2003, 650-662.

[13] S. Sahni, K. Kim, and H. Lu, Data structures for one-dimensional packet classification using most-specific-rule matching, *International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, 2002, 3-14.

[14] H. Song, J. Turner, and J. Lockwood, Shape shifting tries for faster IP route lookup, *Proceedings of 13th IEEE International Conference on Network Protocols*, 2005.

[15] V. Srinivasan and G. Varghese, Faster IP lookups using controlled prefix expansion, *SIGMETRICS*, 1998.

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

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

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

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

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