Prefix- And Interval-Partitioned Dynamic IP Router-Tables *

Haibin Lu Kun Suk Kim Sartaj Sahni
{halu,kskim,sahni}@cise.ufl.edu
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL 32611

Abstract

Two schemes—prefix partitioning and interval partitioning—are proposed to improve the performance of dynamic IP router-table designs. While prefix partitioning applies to all known dynamic router-table designs, interval partitioning applies to the alternative collection of binary search tree designs of Sahni and Kim [16]. Experiments using public-domain IPv4 router databases indicate that one of the proposed prefix partitioning schemes—TLDP—results in router tables that require less memory than when prefix partitioning is not used. Further significant reduction in the time to find the longest matching-prefix, insert a prefix, and delete a prefix is achieved.

Keywords: Packet routing, dynamic router-tables, longest-prefix matching, prefix partitioning, interval partitioning.

1 Introduction

In IP routing, each router table has a set of rules (F, N), where F is a filter and N is the next hop for the packet. Typically, each filter is a destination address prefix and longest-prefix matching is used to determine the next hop for each incoming packet. That is, when a packet arrives at a router, its next hop is determined by the rule that has the longest prefix (i.e., filter) that matches the destination address of the packet. Notice that the length of a router-table prefix cannot exceed the length W of a destination address. In IPv4, destination addresses are $W = 32$ bits long, and in IPv6, $W = 128$.

In a *static* rule table, the rule set does not vary in time. For these tables, we are concerned primarily with the following metrics:

1. *Time required to process an incoming packet.* This is the time required to search the rule table for the rule to use.

2. *Preprocessing time.* This is the time to create the rule-table data structure.

*This work was supported, in part, by the National Science Foundation under grant CCR-9912395.

3. *Storage requirement.* That is, how much memory is required by the rule-table data structure?

In practice, rule tables are seldom truly static. At best, rules may be added to or deleted from the rule table infrequently. Typically, in a "static" rule table, inserts/deletes are batched and the rule-table data structure reconstructed as needed.

In a *dynamic* rule table, rules are added/deleted with some frequency. For such tables, inserts/deletes are not batched. Rather, they are performed in real time. For such tables, we are concerned additionally with the time required to insert/delete a rule. For a dynamic rule table, the initial rule-table data structure is constructed by starting with an empty data structures and then inserting the initial set of rules into the data structure one by one. So, typically, in the case of dynamic tables, the preprocessing metric, mentioned above, is very closely related to the insert time.

Ruiz-Sanchez, Biersack, and Dabbous [12] review data structures for static router-tables and Sahni, Kim, and Lu [18] review data structures for both static and dynamic router-tables. Several trie-based data structures for router table have been proposed [19, 1, 2, 11, 20, 13, 14]. Structures such as that of [19] perform each of the dynamic router-table operations (lookup, insert, delete) in $O(W)$ time. Others (e.g., [1, 2, 11, 20, 13, 14]) attempt to optimize lookup time and memory requirement through an expensive preprocessing step. These structures, while providing very fast lookup capability, have a prohibitive insert/delete time and so, they are suitable only for static router-tables (i.e., tables into/from which no inserts and deletes take place).

Waldvogel et al. [22] have proposed a scheme that performs a binary search on hash tables organized by prefix length. This binary-search scheme has an expected complexity of $O(\log W)$ for lookup. An alternative adaptation of binary search to longest-prefix matching is developed in [7]. Using this adaptation, a lookup in a table that has n prefixes takes $O(W + \log n) = O(W)$ time. Because the schemes of [22] and [7] use expensive precomputation, they are not suited for a dynamic router-tables.

Suri et al. [21] have proposed a B-tree data structure for dynamic router tables. Using their structure, we may find the longest matching-prefix, $lmp(d)$, in $O(\log n)$ time. However, inserts/deletes take $O(W \log n)$ time. When W bits fit in $O(1)$ words (as is the case for IPv4 and IPv6 prefixes) logical

operations on W-bit vectors can be done in $O(1)$ time each. In this case, the scheme of [21] takes $O(\log W \log n)$ time for an insert and $O(W + \log n) = O(W)$ time for a delete. The number of cache misses that occur when the data structure of [21] is used is $O(\log n)$ per opertion. Lu and Sahni [10] have developed an alternative B-tree router-table design. Although the designs of [21] and [10] have the same asymptotic complexity for each of the dynamic router-table operations, inserts and deletes access only $O(\log_m n)$ nodes using the structure of [10] whereas these operations access $O(m \log_m n)$ nodes when the structure of [21] is used. Consequently, the structure of [10] is faster for inserts and deletes and competitive for searches.

Sahni and Kim [15, 16] develop data structures, called a collection of red-black trees (CRBT) and alternative collection of red-black trees (ACRBT), that support the three operations of a dynamic router-table (longest matching-prefix, prefix insert, prefix delete) in $O(\log n)$ time each. The number of cache misses is also $O(\log n)$. In [16], Sahni and Kim show that their ACRBT structure is easily modified to extend the biased-skip-list structure of Ergun et al. [3] so as to obtain a biased-skip-list structure for dynamic router table. Using this modified biased skip-list structure, lookup, insert, and delete can each be done in $O(\log n)$ expected time and $O(\log n)$ expected cache misses. Like the original biased-skip list structure of [3], the modified structure of [16] adapts so as to perform lookups faster for bursty access patterns than for non-bursty patterns. The ACRBT structure may also be adapted to obtain a collection of splay trees structure [16], which performs the three dynamic router-table operations in $O(\log n)$ amortized time and which adapts to provide faster lookups for bursty traffic.

Lu and Sahni [8] use priority search trees to arrive at an $O(\log n)$ data structure for dynamic prefix-tables. This structure is faster than the CRBT structure of [15, 16]. Lu and Sahni [8] also propose a data structure that employs priority search trees and red-black trees for the representation of rule tables in which the filters are a conflict-free set of ranges. This data structure permits most-specific-range matching as well as range insertion and deletion to be done in $O(\log n)$ time each.

In [9], Lu and Sahni develop a data structure called BOB (binary tree on binary tree) for dynamic router-tables in which the rule filters are nonintersecting ranges and in which ties are broken by selecting

the highest-priority rule that matches a destination address. Using BOB, the highest-priority rule that matches a destination address may be found in $O(\log^2 n)$ time; a new rule may be inserted and an old one deleted in $O(\log n)$ time. Related structures PBOB (prefix BOB) and LMPBOB (longest matching-prefix BOB) are proposed for highest-priority prefix matching and longest-matching prefixes. These structures apply when all filters are prefixes. The data structure LMPBOB permits longest-prefix matching in $O(W)$ time; rule insertion and deletion take $O(\log n)$ time each. On practical rule tables, BOB and PBOB perform each of the three dynamic-table operations in $O(\log n)$ time and with $O(\log n)$ cache misses. The number of cache misses incurred by LMPBOB is also $O(\log n)$.

Gupta and McKeown [4] have developed two data structures for dynamic highest-priority rule tables—heap on trie (HOT) and binary search tree on trie (BOT). The HOT structure takes $O(W)$ time for a lookup and $O(W \log n)$ time for an insert or delete. The BOT structure takes $O(W \log n)$ time for a lookup and $O(W)$ time for an insert/delete. The number of cache misses in a HOT and BOT is asymptotically the same as the time complexity of the corresponding operation.

In this paper, we develop two strategies to improve the performance of already known data structures for dynamic router-tables–prefix partitioning (Section 2) and interval partitioning (Section 4). Prefix partitioning is quite general and may be used in conjunction with all known dynamic IP router-table designs. However, interval partitioning applies only to the interval-based designs of Sahni and Kim [16]. Experimental results are presented in Section 5.

2 Prefix Partitioning

2.1 Static Router-Tables

Lampson et al.[7] propose a prefix partitioning scheme for static router-tables. This scheme partitions the prefixes in a router table based on their first s, $s \leq W$, 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 suitable for a static router-table. Further, $partition[i].lmp$, which is the longest prefix 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 Q equal null if no prefix in $partition[i].bucket$ matches d; otherwise, let Q be the longest prefix in $partition[i].bucket$ that matches d.

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

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. Both the average-case and worst-case time to find $lmp(d)$ decrease as we increase s. However, the storage needed for the array $partition[]$ increases with s and quickly becomes impractical. Lampson et al. [7] recommend using $s = 16$. This recommendation results in $2^s = 65{,}536$ buckets. For practical router-table databases, $s = 16$ results in buckets that have at most a few hundred prefixes; non-empty buckets have less than 10 prefixes on average. Hence the worst-case and average-case time to find $lmp(d)$ is considerably improved over the case $s = 0$.

2.2 Dynamic Router-Tables

The prefix partitioning scheme of Lampson et al. [7] is, however, not suited for dynamic router-tables. This is so because the insertion or deletion of a prefix may affect all $partition[i].lmp$, $0 \leq i < 2^s$ values. For example, the insertion of the default prefix * into an initially empty router table will require all 2^s $partition[i].lmp$ values to be set to *. The deletion of a length $p \leq s$ prefix could affect 2^{s-p} $partition[i].lmp$ values.

For dynamic router-tables, we propose multilevel partitioning. The prefixes at each node of the partitioning tree are partitioned into $2^s + 1$ partitions using the next s bits of the prefixes. Prefixes that do not have s additional bits fall into partition -1; the remaining prefixes fall into the partition that corresponds to their next s bits. Prefix partitioning may be controlled using a static rule such as "partition only at the root" or by a dynamic rule such as "recursively partition until the number of prefixes in the partition is smaller than some specified threshold". In this paper, we focus on two statically determined partitioning structures—one level and two level.

2.2.1 One-Level Dynamic Partitioning (OLDP)

OLDP is described by the partitioning tree of Figure 1. The root node represents the partitioning of the router-table prefixes into $2^s + 1$ partitions. Let $OLDP[i]$ refer to partition i. $OLDP[-1]$ contains all prefixes whose length is less than s; $OLDP[i]$, $i \geq 0$ contains all prefixes whose first s bits correspond to i. The prefixes in each partition may be represented using any of the dynamic router-table data structures mentioned in Section 1. In fact, one could use different data structures for different partitions. For example, if we knew that certain partitions will always be small, these could be represented as linear lists while the remaining partitions are represented using PBOB (say).

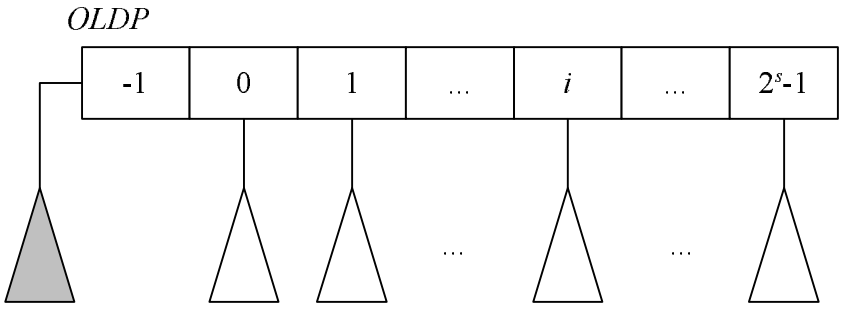

Figure 1: One-level dynamic partitioning

The essential difference between OLDP and the partitioning scheme of [7] is in the treatment of prefixes whose length is smaller than s. In OLDP, these shorter prefixes are stored in $OLDP[-1]$; in the scheme of [7] these shorter prefixes (along with length s prefixes) are used to determine the $partition[i].lmp$ values. It is this difference in the way shorter length prefixes are handled that makes OLDP suitable for dynamic tables while the scheme of [7] is suitable for static tables. Table 1 gives the results of partitioning four IPv4 router tables using OLDP with $s = 16$ and Figure 2 histograms the number of partitions (excluding partition -1) of each size. These router tables were obtained from [6]. As can be seen, OLDP with $s = 16$ is quite effective in reducing both the maximum and the average partition size. In all of our databases, partition -1 is substantially larger than the remaining partitions.

Figures 3–5 give the algorithms to search, insert, and delete into an OLDP router table. `OLDP[i]->x()` refers to method `x` performed on the data structure for `OLDP[i]`, $first(d, s)$ returns the integer that

Database	Paix	Pb	Aads	MaeWest
# of prefixes	85988	35303	31828	28890
# of nonempty partitions	10443	6111	6363	6026
$\|OLDP[-1]\|$	586	187	188	268
Max size of remaining partitions	229	124	112	105
Average size of nonempty partitions (excluding $OLDP[-1]$)	8.2	5.7	5.0	4.8

Table 1: Statistics of one level partition ($s = 16$)

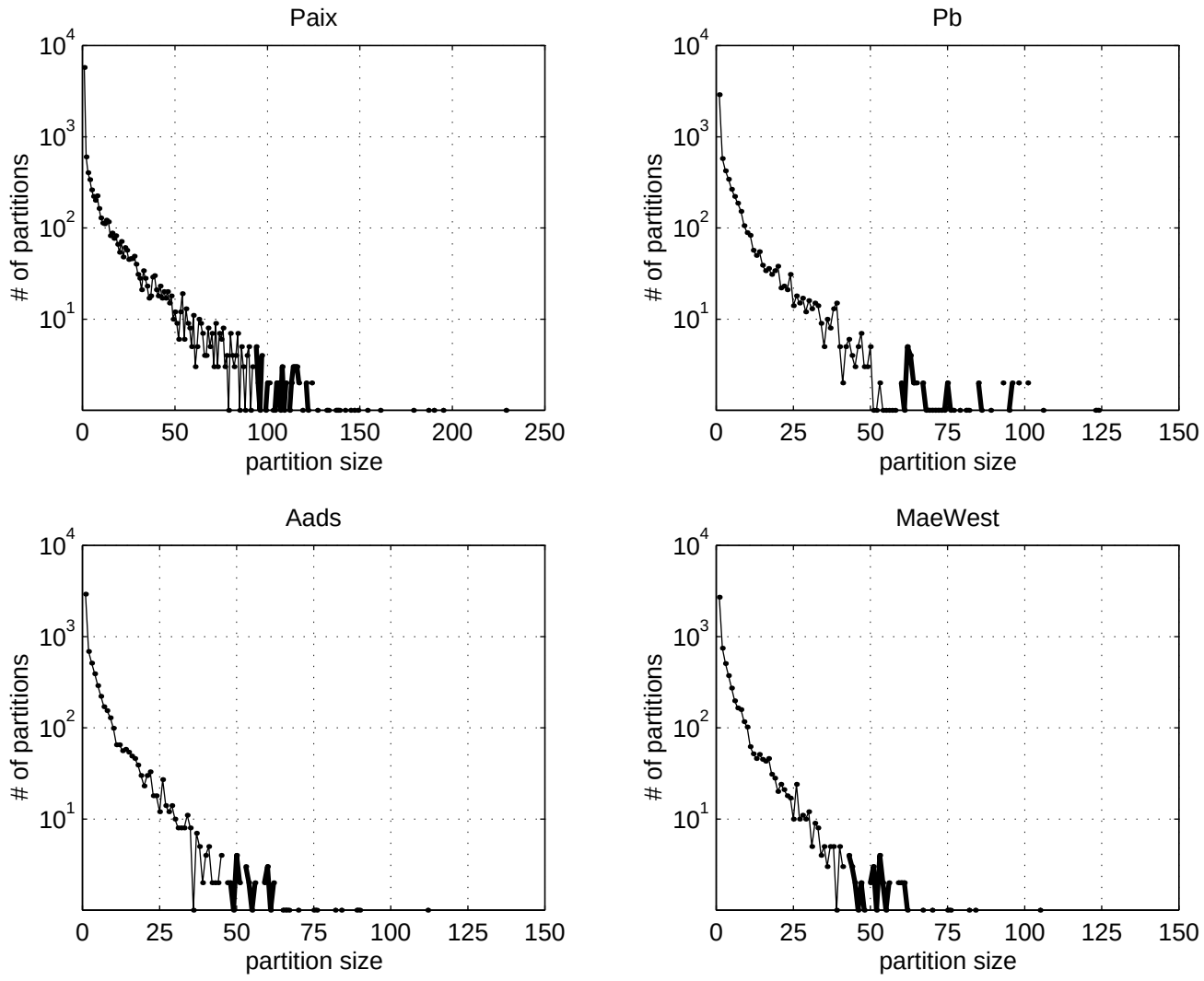

Figure 2: Histogram of partition size for OLDP (excludes $OLDP[-1]$)

corresponds to the first s bits of d, and $length(thePrefix)$ returns the length of $thePrefix$. It is easy to see that when each OLDP partition is represented using the same data structure (say, PBOB), the

```
Algorithm lookup(d){
   // return the lmp(d)
   if(OLDP[first(d, s)] != null && lmp = OLDP[first(d, s)]->lookup(d)){
      // found matching prefix in OLDP[first(d, s)]
      return lmp;
   }
   return OLDP[-1]->lookup(d);
}
```

Figure 3: OLDP algorithm to find $lmp(d)$

```
Algorithm insert(thePrefix){
   // insert prefix thePrefix
   if(length(thePrefix) >= s)
      OLDP[first(thePrefix, s)]->insert(thePrefix);
   else
      OLDP[-1]->insert(thePrefix);
}
```

Figure 4: OLDP algorithm to insert a prefix

```
Algorithm delete(thePrefix){
   // delete prefix thePrefix
   if (length(thePrefix) >= s)
      OLDP[first(thePrefix, s)]->delete(thePrefix);
   else
      OLDP[-1]->delete(thePrefix);
}
```

Figure 5: OLDP algorithm to delete a prefix

asymptotic complexity of each operation is the same as that for the corresponding operation in the data structure for the OLDP partitions. However, a constant factor speedup is expected because each $OLDP[i]$ has only a fraction of the prefixes.

2.2.2 Two-Level Dynamic Partitioning (TLDP)

Figure 6 shows the partitioning structure for a TLDP. In a TLDP, the root partitions the prefix set into the partitions $OLDP[i]$, $-1 \leq i < 2^s$ by using the first s bits of each prefix. This partitioning is identical to that done in an OLDP. Additionally, the set of prefixes $OLDP[-1]$ is further partitioned at node $TLDP$ into the partitions $TLDP[i]$, $-1 \leq i < 2^t$ using the first t, $t < s$, bits of the prefixes in

$OLDP[-1]$. This partitioning follows the strategy used at the root. However, t rather than s bits are used. The prefix partitions $OLDP[i]$, $0 \leq i < 2^s$ and $TLDP[i]$, $-1 \leq i < 2^t$, may be represented using any dynamic router-table data structure. Note that the $OLDP[i]$ partitions for $i \geq 0$ are not partitioned further because their size is typically not too large (see Table 1).

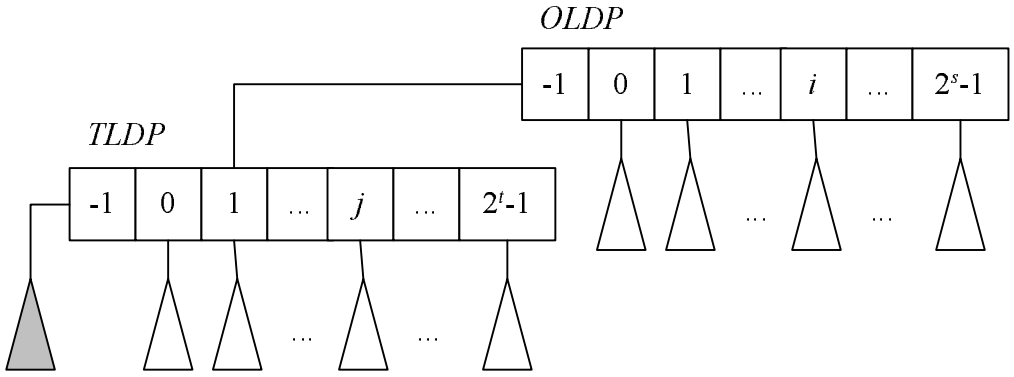

Figure 6: Two-level dynamic partitioning

Table 2 gives statistics for the number of prefixes in $TLDP[i]$ for our four sample databases. Since the number of prefixes in each $TLDP[i]$ is rather small, we may represent each $TLDP[i]$ using an array linear list in which the prefixes are in decreasing order of length.

Database	Paix	Pb	Aads	MaeWest
$\|OLDP[-1]\|$	586	187	188	268
# of nonempty TLDP partitions	91	57	53	67
$\|TLDP[-1]\|$	0	0	0	0
$\max\{\|TLDP[i]\|\}$	33	12	15	15
Average size of nonempty TLDP partitions	6.4	3.3	3.5	4.0

Table 2: Statistics for TLDP with $s = 16$ and $t = 8$

Figures 7 and 8 give the TLDP search and insert algorithms. The algorithm to delete is similar to the insert algorithm. It is easy to see that TLDP doesn't improve the asymptotic complexity of the lookup/insert/delete algorithms relative to that of these operations in an OLDP. Rather, a constant factor improvement is expected.

```
Algorithm lookup(d){
   // return lmp(d)
   if (OLDP[first(d, s)] != null && lmp = OLDP[first(d, s)]->lookup(d)){
      // found lmp in OLDP[first(d, s)]
      return lmp;
   }
   if (TLDP[first(d, t)] != null && lmp = TLDP[first(d, t)]->lookup(d)){
      // found lmp in TLDP[first(d, t)]
      return lmp;
   }
   return TLDP[-1]->lookup(d);
}
```

Figure 7: TLDP algorithm to find $lmp(d)$

```
Algorithm insert(thePrefix){
   // insert prefix thePrefix
   if (length(thePrefix) >= s)
      OLDP[first(thePrefix, s)]->insert(thePrefix);
   else if (length(thePrefix) >= t)
      TLDP[first(thePrefix, t)]->insert(thePrefix);
   else
      TLDP[-1]->insert(thePrefix);
}
```

Figure 8: TLDP algorithm to insert a prefix

2.2.3 Extension to IPv6

Although OLDPs with $s = 16$ and TLDPs with $s = 16$ and $t = 8$ seem quite reasonable for IPv4 router tables, for IPv6 router tables, we expect better performance using OLDPs with $s = 64$ (say) and TLDPs with $s = 64$ and $t = 32$ (say). However, maintaining the $OLDP$ and $TLDP$ nodes as arrays as in Sections 2.2.1 and 2.2.2 is no longer practical (e.g., the OLDP array will have $2^s = 2^{64}$ entries). Notice, however, that since most of the the OLDP and TLDP partitions are expected to be empty (even in IPv6)[1], the $OLDP$ and $TLDP$ nodes may be efficiently represented as hash tables. *A similarly memory-efficient hash table representation of the partition array of the scheme of [7] isn't possible because virtually all of the partition*$[i]$*.lmp values are non-null.*

[1]Note that the number of non-empty partitions is $O(n)$.

3 OLDP and TLDP Fixed-Stride Tries

3.1 Fixed-Stride Tries

A trie node whose stride is s has 2^s subtries, some or all of which may be empty. A fixed-stride trie (FST) [18, 20] is a trie in which all nodes that are at the same level have the same stride. The nodes at level i of an FST store prefixes whose length, $length(i)$, is $\sum_{j=0}^{i} s_j$, where s_j is the stride for nodes at level j. Suppose we wish to represent the prefixes of Figure 9(a) using an FST that has three levels. Assume that the strides are 3, 2, and 2. The root of the trie stores prefixes whose length is 3; the level one nodes store prefixes whose length is 5 (3 + 2); and the level two nodes store prefixes whose length is 7 (3 + 2 + 2). This poses a problem for the prefixes of our example, because the length of some of these prefixes is different from the storeable lengths. For instance, the length of P3 is 2. To get around this problem, a prefix with a nonpermissible length is expanded to the next permissible length [20]. For example, P3 = 11* is expanded to P3a = 110* and P3b = 111*. If one of the newly created prefixes is a duplicate, natural dominance rules are used to eliminate all but one occurrence of the prefix. For instance, P7 = 110000* is expanded to P7a = 1100000* and P7b = 1100001*. However, P8 = 1100000* is to be chosen over P7a = 1100000*, because P8 is a longer match than P7. So, P7a is eliminated. Because of the elimination of duplicate prefixes from the expanded prefix set, all prefixes are distinct. Figure 9(b) shows the prefixes that result when we expand the prefixes of Figure 9(a) to lengths 3, 5, and 7. Figure 10 shows the corresponding FST whose height is 2 and whose strides are 3, 2, and 2.

Since the trie of Figure 10 can be searched with at most 3 memory references, it represents a time performance improvement over a 1-bit trie (this is an FST in which the stride at each level is 1), which requires up to 7 memory references to perform a search for our example prefix set. For any given set of prefixes, the memory required by an FST of whose height is at most k depends on the strides of the up to $k+1$ levels in the FST. [20, 13] develop efficient algorithms to find the up to $k+1$ strides that result in the most memory efficient FSTs. For dynamic router-tables, however, the optimal strides change with each insert and delete operation. So, instead of maintaining optimality of strides dynamically, we must fix the strides based on expected characteristics of the prefix set. The use of expected characteristics

$P1 = 0*$	$000 * (P1a)$
$P2 = 1*$	$001 * (P1b)$
$P3 = 11*$	$010 * (P1c)$
$P4 = 101*$	$011 * (P1d)$
$P5 = 10001*$	$100 * (P2)$
$P6 = 1100*$	$101 * (P4)$
$P7 = 110000*$	$110 * (P3a)$
$P8 = 1100000*$	$111 * (P3b)$
	$10001 * (P5)$
	$11000 * (P6a)$
	$11001 * (P6b)$
	$1100000 * (P8)$
	$1100001 * (P7)$
(a) Original prefixes	(b) Expanded prefixes

Figure 9: A prefix set and its expansion to three lengths

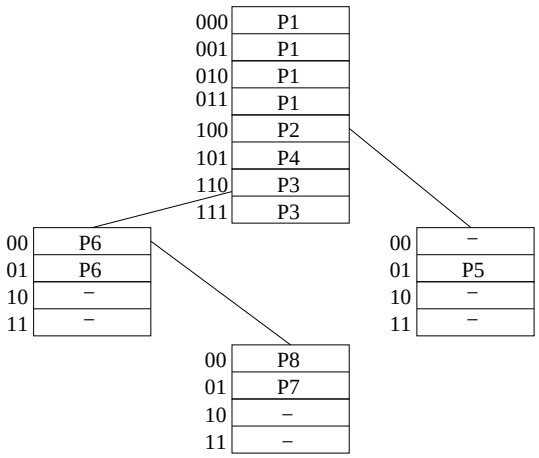

Figure 10: FST for expanded prefixes of Figure 9(b)

precludes the use of variable-stride tries [20, 14].

To determine the strides of the FST for dynamic tables, we examine the distribution of prefixes in our Paix database (Table 3). Fewer than 0.7% of the Paix prefixes have length < 16. Hence using a root stride of 16 will require us to expand only a small percentage of the prefixes from length < 16 to length 16. Using a larger stride for the root will require us to expand the 6606 prefixes of length 16. So, we set the root stride at 16. For the children and grandchildren of the root, we choose a stride of 4. This choice requires us to expand prefixes whose length is 17, 18, and 19 to length 20 and expand prefixes of length

Len	Num of prefixes	% of prefixes	Cum % of prefixes	Len	Num of prefixes	% of prefixes	Cum % of prefixes
1	0	0.0000	0.0000	17	918	1.0714	9.4652
2	0	0.0000	0.0000	18	1787	2.0856	11.5509
3	0	0.0000	0.0000	19	5862	6.8416	18.3924
4	0	0.0000	0.0000	20	3614	4.2179	22.6103
5	0	0.0000	0.0000	21	3750	4.3766	26.9870
6	0	0.0000	0.0000	22	5525	6.4483	33.4353
7	0	0.0000	0.0000	23	7217	8.4230	41.8583
8	22	0.0267	0.0257	24	49756	58.0705	99.9288
9	4	0.0047	0.0303	25	12	0.0140	99.9428
10	5	0.0058	0.0362	26	26	0.0303	99.9732
11	9	0.0105	0.0467	27	12	0.0140	99.9872
12	26	0.0303	0.0770	28	5	0.0058	99.9930
13	56	0.0654	0.1424	29	4	0.0047	99.9977
14	176	0.2054	0.3478	30	1	0.0012	99.9988
15	288	0.3361	0.6839	31	0	0.0000	99.9988
16	6606	7.7099	8.3938	32	1	0.0012	100.0000

Table 3: Distribution of prefixes in Paix

21, 22, and 23 to length 24. The level 4 nodes may be given a stride of 8, requiring the expansion of the very few prefixes whose length is between 25 and 31 to a length of 32. These stride choices result in a 16-4-4-8-FST (root stride is 16, level 1 and level 2 stride is 4, level 3 stride is 8). Since a 16-4-4-8-FST has 4 levels, $lmp(d)$ may be found with at most 4 memory accesses. Other reasonable stride choices result in a 16-8-8-FST and a 16-4-4-4-4-FST.

FST Operations

To find $lmp(d)$ using an FST, we simply use the bits of d to follow a path from the root of the FST toward a leaf. The last prefix encountered along this path is $lmp(d)$. For example, to determine $lmp(1100010)$ from the 3-2-2-FST of Figure 10, we use the first 3 bits (110) to get to the left level 1 node. The next 2 bits 00 are used to reach the level 2 node. Finally, using the last 2 bits 10, we fall off the trie. The prefixes encountered on this path are P3 (in the root) and P6 (in the level 1 node; node that no prefix is encountered in the 10 field of the level 2 node). The last prefix encountered is P6. Hence, $lmp(1100010)$ is P6.

To insert the prefix p, we follow a search path determined by the bits of p until we reach the level i node

N with the property $length(i-1) < length(p) \leq length(i)$ (for convenience, assume that $length(-1) = 0$; note that it may be necessary to add empty nodes to the trie in case such an N isn't already in the trie). The prefix p is expanded to length $length(i)$ and stored in the node slots for each of the expanded prefixes (if a slot is already occupied, p is stored in the occupied slot only if it is longer than the prefix occupying that spot).

To facilitate the delete operation, each node M of an FST maintains an auxilliary Boolean array $M.prefixes[0 : 2^s - 1]$, where s is the stride of M. This array keeps track of the prefixes inserted at node M. When prefix p is inserted, $N.prefixes[q]$ is set to true. Here N is as described above and q is determined as follows. Let $number(i, p)$ be the number represented by bits $length(i-1) \cdots length(p) - 1$ of p (the bits of p are indexed from left to right beginning with the index 0). So, for example, the bit-sequence 010 represents the number 2. q equals $2^{length(p)-length(i-1)} + number(i, p) - 2$. An alternative to the array $M.prefixes[]$ is to keep track of the prefixes inserted at node M using a trie on bits $length(i-1)\cdots$ of the inserted prefixes. Since our implementation doesn't use this alternative, we do not consider the alternative further.

To delete the prefix p, we find the node N as for an insert operation. $N.prefixes[q]$ is set to false, where q is computed as above. To update the prefix slots of N that contain p, we need to find the longest proper prefix of p that is in $N.prefixes$. This longest proper prefix is determined by examining $N.prefixes[j]$ for $j = 2^{r-length(i-1)} + number(i, p_r) - 2$, $r = length(p) - 1$, $length(p) - 2$, $\cdots$, $length(i-1) + 1$, where p_r is the first r bits of p. The examination stops at the first j for which $N.prefixes[j]$ is true. The corrsponding prefix replaces p in the prefix slots of N. If there is no such j, the null prefix replaces p.

3.2 OLDP and TLDP FSTs

Since the root stride is 16 for the recommended IPv4 FSTs (16-4-4-8, 16-4-4-4-4, and 16-8-8) of Section 3.1 and since $s = 16$ is recommended for IPv4, an OLDP 16-4-4-4-4-FST (for example) has the structure shown in Figure 1 with each $OLDP[i]$, $i \geq 0$ being a 4-4-4-4-FST; $OLDP[-1]$ is a 4-4-4-3-FST. The root of each 4-4-4-4-FST, while having a stride of 4 needs to account for prefixes of length 16 through 20. A TLDP 16-4-4-4-4-FST has the structure of Figure 6 with each $OLDP[i]$, $i \geq 0$ being a 4-4-4-4-FST; each

Prefix Name	Prefix	Range Start	Range Finish
P1	*	0	31
P2	0101*	10	11
P3	100*	16	19
P4	1001*	18	19
P5	10111	23	23

Table 4: Prefixes and their ranges, $W = 5$

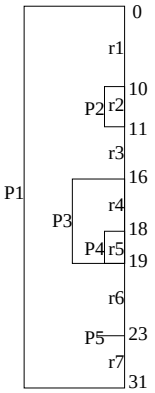

Figure 11: Pictorial representation of prefixes and ranges

$TLDP[i]$, $i \geq 0$ is a 4-3-FST; and $TLDP[-1]$ is a 4-3-FST. The root of each $TLDP[i]$ 4-3-FST, while having a stride of 4 needs to account for prefixes of length 8 through 12.

4 Interval Partitioning

Table 4 gives a set of 5 prefixes along with the range of destination addreses matched by each prefix. This table assumes that $W = 5$. Figure 11 gives the pictorial representation of the five prefixes of Table 4. The end points of the 5 prefixes of Table 4 are (in ascending order) 0, 10, 11, 16, 18, 19, 23, and 31. Two consecutive end points define a *basic interval*. So, the basic intervals defined by our 5 prefixes are $r1 = [0, 10]$, $r2 = [10, 11]$, $r3 = [11, 16]$, $r4 = [16, 18]$, $r5 = [18, 19]$, $r6 = [19, 23]$, and $r7 = [23, 31]$. In general, n prefixes may have up to $2n$ distinct end points and $2n - 1$ basic intervals.

For each prefix and basic interval, x, define $next(x)$ to be the smallest range prefix (i.e., the longest prefix) whose range includes the range of x. For the example of Figure 11, the $next()$ values for the basic intervals r1 through r7 are, respectively, P1, P2, P1, P3, P4, P1, and P1.

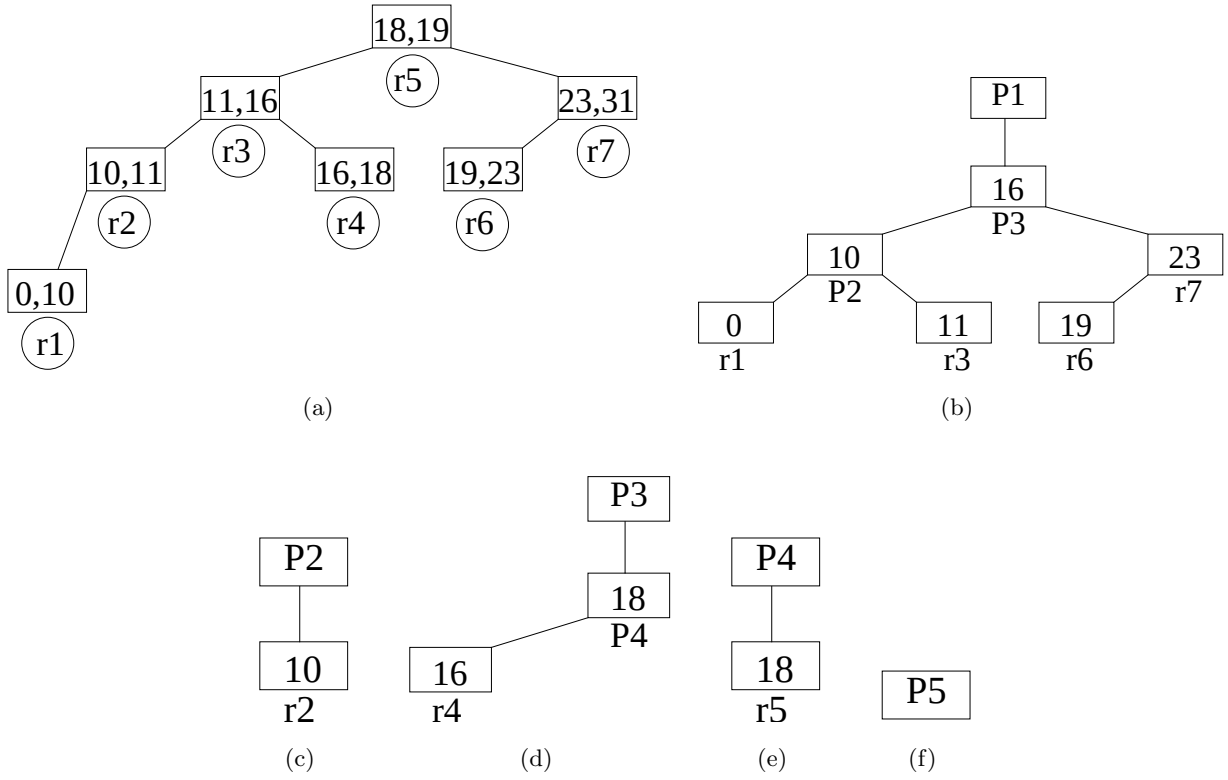

Figure 12: ACBST of [16]. (a) Alternative basic interval tree (b) prefix tree for P1 (c) prefix tree for P2 (d) prefix tree for P3 (e) prefix tree for P4 (f) prefix tree for P5

The dynamic router-table structures of Sahni and Kim [15, 16] employ a front-end basic-interval tree (BIT) that is used to determine the basic interval that any destination address d falls in. The back-end structure, which is a collection of prefix trees (CPT), has one prefix tree for each of the prefixes in the router table. The prefix tree for prefix P comprises a header node plus one node, called a *prefix node*, for every nontrivial prefix (i.e., a prefix whose start and end points are different) or basic interval x such that $next(x) = P$. The header node identifies the prefix P for which this is the prefix tree. The BIT as well as the prefix trees are binary search trees.

Figure 12(a) shows the BIT (actually, alternative BIT, ABIT) for our 5-prefix example and Figures 12(b)-(f) show the back-end prefix trees for our 5 prefixes. Each ABIT node stores a basic interval. Along with each basic interval, a pointer to the back-end prefix-tree node for this basic interval is stored. Additionally, for the end points of this basic interval that correspond to prefixes whose length is W,

a pointer to the corresponding W-length prefixes is also stored. In Figure 12(a), the end point prefix pointers for the end point 23 are not shown; remaining end point prefix pointers are null; the pointers to prefix-tree nodes are shown in the circle outside each node.

In Figures 12(b)-(f), notice that prefix nodes of a prefix tree store the start point of the range or prefix represented by that prefix node. The start points of the basic intervals and prefixes are shown inside the prefix nodes while the basic interval or prefix name is shown outside the node.

To find $lmp(9)$, we use the ABIT to reach the ABIT node for the containing basic interval $[0, 10]$. This ABIT node points us to node $r1$ in the back-end tree for prefix P1. Following parent pointers from node $r1$ in the back-end tree, we reach the header node for the prefix tree and determine that $lmp(9)$ is P1. When determining $lmp(16)$, we reach the node for $[16, 18]$ and use the pointer to the basic interval node $r4$. Following parent pointers from $r4$, we reach the header node for the prefix tree and determine that $lmp(16)$ is P3. To determine $lmp(23)$, we first get to the node for $[23, 31]$. Since this node has a pointer for the end point 23, we follow this pointer to the header node for the preifx tree for P5, which is $lmp(23)$.

The *interval partitioning* scheme is an alternative to the OLDP and TLDP partitioning schemes that may be applied to interval-based structures such as the ACBST. In this scheme, we employ a 2^s-entry table, *partition*, to partition the basic intervals based on the first s bits of the start point of each basic interval. For each partition of the basic intervals, a separate ABIT is constructed; the back-end prefix trees are not affected by the partitioning. Figure 13 gives the partitioning table and ABITs for our 5-prefix example and $s = 3$.

Notice that each entry $partition[i]$ of the partition table has four fields– *abit* (pointer to ABIT for partition i), *next* (next nonempty partition), *previous* (previous nonempty partition), and *start* (smallest end point in partition). Figure 14 gives the interval partitioning algorithm to find $lmp(d)$. The algorithm assumes that the default prefix * is always present, and the method `rightmost` returns the *lmp* for the rightmost basic interval in the ABIT.

The insertion and deletion of prefixes is done by inserting and removing end points, when necessary,

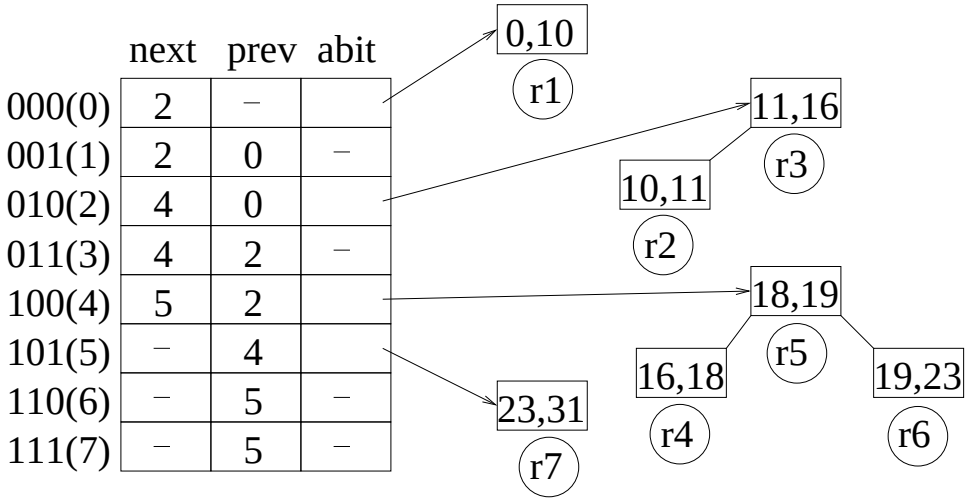

Figure 13: Interval-partitioned ABIT structures corresponding to Figure 12

```
Algorithm lookup(d){
   // return lmp(d)
   p = first(d,s);
   if (partition[p].abit != null && partition[p].start <= d)
      // containing basic interval is in partition[p].abit
      return partition[p].abit->lookup(d);
   else return partition[partition[p].previous].abit->rightmost();
}
```

Figure 14: Interval partitioning algorithm to find $lmp(d)$

from the ABIT and adding/removing a back-end prefix tree. The use of interval partitioning affects only the components of the insert/delete algorithms that deal with the ABIT. Figures 15 and 16 give the algorithms to insert and delete an end point. `rightPrefix` refers to the prefix, if any, associated with the right end point stored in a node of the ABIT.

Although the application of interval partitioning doesn't change the asymptotic complexity, $O(\log n)$, of the ACBST algorithm to find the longest matching-prefix, the complexity of the algorithms to insert and delete change from $O(\log n)$ to $O(\log n + 2^s)$. Since interval partitioning reduces the size of individual ABITs, a reduction in observed runtime is expected for the search algorithm. The insert and delete algorithms are expected to take less time when the clusters of empty partitions are relatively small (equivalently, when the non-empty partitions distribute uniformly across the partition table).

```
Algorithm insert(e){
   // insert the end point e
   p = first(e,s);
   if (partition[p].abit == null){
      // rn has interval that is split by e
      rn = partition[partition[p].previous].abit->rightmostNode();
      // split into 2 intervals
      partition[p].abit->new Node(e, rn->rightKey, rn->rightPrefix);
      rn->rightKey = e; rn->rightPrefix = null;
      // update next and previous fields of the partition table
      for (i=partition[p].previous; i<p; i++) partiton[i].next = p;
      for (i=p+1; i<=partition[p].next; i++) partiton[i].previous = p;
   }
   else{
      if (partition[p].start > e){
         rn = partition[partition[p].previous].abit->rightmostNode();
         rn->rightKey = e; rn->rightPrefix = null;
      }
      partition[p].abit->insert(e);
   }
}
```

Figure 15: Interval partitioning algorithm to insert an end point

```
Algorithm delete(e){
   // delete the end point e
   p = first(e,s);
   if (partition[p].abit != null){
      if (partition[p].start == e){
         // combine leftmost interval of p with rightmost of previous
         ln = partition[p].abit->leftmostNode();
         rn = partition[partition[p].previous].abit->rightmostNode();
         rn->rightKey = ln->rightKey; rn->rightPrefix = ln->rightPrefix;
      }
      partition[p].abit->delete(e);
      if (partition[p].abit == null){
         // update next and previous fields of the partition table
         for (i=partition[p].previous; i<p; i++)
             partiton[i].next = partiton[p].next;
         for (i=p+1; i<=partition[p].next; i++)
             partiton[i].previous = partiton[p].previous;
      }
   }
}
```

Figure 16: Interval partitioning algorithm to delete an end point

5 Experimental Results

To assess the efficacy of the proposed prefix- and interval-partitioning schemes, we programmed these schemes in C++. For prefix-partitioning, we experimented with using the following dynamic router-table structures as the $OLDP[i]$, $i \geq 0$ structure (as well as the $OLDP[-1]$ structure in case of one-level dynamic partitioning): ACRBT (the ACBST of [16] with each search tree being a red-black tree), CST (the ACBST of [16] with each search tree being a splay tree), MULTIBIT (16-4-4-4-4 FST; in OLDP applications, 4-4-4-4-FSTs are used for $OLDP[i]$, $i \geq 0$ and a 4-4-4-3-FST is used for $OLDP[-1]$; in TLDP applications, 4-4-4-4-FSTs are used for $OLDP[i]$, $i \geq 0$, 4-3-FSTs for $TLDP[i]$, $i \geq 0$, and a 4-3-FST for $TLDP[-1]$), MULTIBITb (16-8-8 FST; in OLDP applications, 8-8-FSTs are used for $OLDP[i]$, $i \geq 0$ and an 8-7-FST is used for $OLDP[-1]$; in TLDP applications, 8-8 FSTs are used for $OLDP[i]$, $i \geq 0$, 8-FSTs for $TLDP[i]$, $i \geq 0$, and a 7-FST is used for $TLDP[-1]$), PST (the prefix search trees of [8]), PBOB (the prefix binary tree on binary tree structure of [9]), TRIE (one-bit trie) and ARRAY (this is an array linear list in which the prefixes are stored in a one-dimensional array in non-decreasing order of prefix length; the longest matching-prefix is determined by examining the prefixes in the order in which they are stored in the one-dimensional array; array doubling is used to increase array size, as necessary, during an insertion).

We use the notation ACRBT1p (ACRBT1 pure), for example, to refer to OLDP with ACRBTs. ACRBT2p refers to TLDP with ACRBTs. ACRBTIP refers to interval partitioning applied to ACRBTs and CSTIP refers to interval partitioning applied to CSTs.

The schemes whose name end with an "a" (for example, ACRBT2a) are variants of the corresponding pure schemes. In ACRBT2a, for example, each of the TLDP codes, $TLDP[i]$ was implemented as an array linear list until $|TLDP[i]| > \tau$, where the threshold τ was set to 8. When $|TLDP[i]| > \tau$ for the first time, $TLDP[i]$ was transformed from an array linear list to the target dynamic router-table structure (e.g., PBOB in the case of PBOB2). Once a $TLDP[i]$ was transformed into the target dynamic router-table structure, it was never transformed back to the array linear list structure no matter how small $|TLDP[i]|$ became. Similarly, $OLDP[i]$, $i \geq 0$ for TLDPs were implemented as array linear lists

until $|OLDP[i]| > \tau$ for the first time. A similar use of array linear lists was made when implementing the OLDP codes.

Note that when $\tau = 0$, we get the corresponding pure scheme (i.e., when $\tau = 0$, ACRBT1a is equivalent to ACRBT1p and PBOB2a is equivalent to PBOB2p, for example) and when $\tau = \infty$, we get one of the two partitioned ARRAY schemes (i.e., ACRBT1a, CST1a, PST1a, PBOB1a, etc. are equivalent to ARRAY1p while ACRBT2a, CST2a, MULTIBIT2a, etc. are equivalent to ARRAY2p). By varying the threshold τ between the two extremes 0 and ∞ the performance of hybrid schemes such as ACRBT1a, MULTIBIT2a, etc. can be varied between that of a pure partitioned scheme and that of ARRAY1p and ARRAY2p.

ACRBT2aH refers to ACRBT2a in which the root-level partitioning node is represented using a hash table rather than an array. The remaining acronymns used by us are easy to figure out. For the OLDP and interval partitioning schemes, we used $s = 16$ and for the TLDP schemes, we used $s = 16$ and $t = 8$. Note that the combinations ARRAY1a and ARRAY2a are the same as ARRAY1p and ARRAY2p. Hence, ARRAY1a and ARRAY2a do not show up in our tables and figures.

Our codes were run on a 2.26GHz Pentium 4 PC that has 500MB of memory. The Microsoft Visual C++ 6.0 compiler with optimization level -O2 was used. For test data, we used the four IPv4 prefix databases of Table 1.

Total Memory Requirement

Tables 5 and 6 and Figure 17 show the amount of memory used by each of the tested structures[2]. In the figure, OLDPp refers to the pure one-level dynamic prefix partitioning versions of the base schemes and INTP refers to the interval partitioning versions. Notice that the amount of memory required by a base data structure (such as ACRBT) is generally less than that required by its OLDP version (ACRBT1p and ACRBT1a) and by its interval partitioning version (where applicable). ACRBT1a, ACRBT1p, CST1p, ACRBTIP, and CSTIP with Paix are the some of the exceptions. In the case of MaeWest, for example, the memory required by PBOB1p is about 39% more than that required by PBOB. The

[2]We did not experiment with the base ARRAY structure, because its run time performance, $O(n)$, is very poor on databases as large as our test databases. As we shall see later, the measured performance of partitioned structures that use ARRAY as a base structure is very good.

TLDP structures (both with an array for the $OLDP$ node and with a hash table for this node) took considerably less memory than did the corresponding base structure. For example, MULTIBIT2a with MaeWest required only 45% of the memory taken by MULTIBIT and MULTIBITb2a with MaeWest took 23% of the memory taken by MULTIBITb. So, although the partitioning schemes were designed so as to reduce run time, the TLDPa schemes also reduce memory requirement! Of the tested structures, ARRAY1p and ARRAY2p are the most memory efficient. However, since the worst-case time to search, insert, and delete in these structures is $O(n)$ (in practice, the times are quite good, because the prefixes in our test databases distribute quite well and the size of each $OLDP[i]$ and $TLDP[i]$ is quite small), we focus also on the best from among the structures that guarantee a good worst-case performance. Of these latter structures, PBOB is the most memory efficient. On the Paix database, for example, PBOB1a takes only 19% of the memory taken by ACRBT1a and only 79% of the memory taken by TRIE1a; PBOB takes 16% of the memory taken by ACRBT and 75% of the memory taken by TRIE.

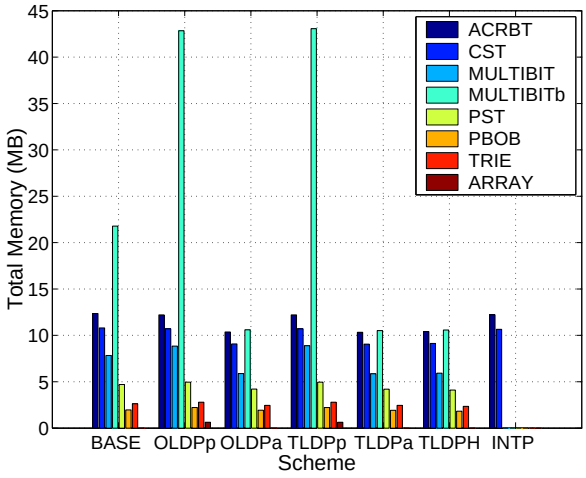

Figure 17: Total memory requirement (in MB) for Paix

Search Time

To measure the average search time, we first constructed the data structure for each of our four prefix databases. Four sets of test data were used. The destination addresses in the first set, NONTRACE, comprised the end points of the prefixes corresponding to the database being searched. These end points

Scheme	Paix	Pb	Aads	MaeWest
ACRBT	12360	5102	4587	4150
CST	10800	4457	4008	3636
MULTIBIT	7830	4939	4982	4685
MULTIBITb	21778	16729	19177	17953
PST	4702	1930	1740	1579
PBOB	1961	811	729	661
TRIE	2622	1091	980	890
ACRBT1p	12211	5479	5023	4638
CST1p	10708	4846	4451	4115
MULTIBIT1p	8848	5087	5098	4731
MULTIBITb1p	42845	25368	27278	24920
PST1p	4958	2186	1996	1835
PBOB1p	2219	1067	985	917
TRIE1p	2799	1279	1163	1075
ARRAY1p	632	417	405	392
ACRBT1a	10361	3736	3151	2787
CST1a	9079	3306	2799	2481
MULTIBIT1a	5884	2644	2439	2119
MULTIBITb1a	10605	4862	5588	4183
PST1a	4209	1603	1377	1237
PBOB1a	1928	851	757	697
TRIE1a	2457	1021	893	815
ACRBT2p	12212	5482	5027	4641
CST2p	10711	4849	4455	4119
MULTIBIT2p	8891	5104	5113	4752
MULTIBITb2p	43068	25503	27391	25065
PST2p	4959	2187	1997	1836
PBOB2p	2220	1068	986	918
TRIE2p	2799	1279	1163	1075
ARRAY2p	634	418	406	393
ACRBT2a	10337	3720	3137	2767
CST2a	9060	3292	2786	2463
MULTIBIT2a	5858	2621	2414	2088
MULTIBITb2a	10514	4778	5498	4075
PST2a	4201	1597	1372	1229
PBOB2a	1926	850	755	695
TRIE2a	2452	1018	890	811
ACRBTIP	12218	5344	4856	4453
CSTIP	10658	4699	4277	3928

Table 5: Memory requirement (in KB)

Scheme	Paix	Pb	Aads	MaeWest
ACRBT2aH	10407	3656	3080	2699
CST2aH	9130	3228	2730	2397
MULTIBIT2aH	5928	2557	2359	2022
MULTIBITb2aH	10584	4714	5443	4009
PST2aH	4101	1498	1272	1129
PBOB2aH	1826	750	656	595
TRIE2aH	2353	919	790	711

Table 6: Memory requirement (in KB) (hash schemes)

were randomly permuted. The data set, PSEUDOTRACE, was constructed from NONTRACE by selecting 1000 destination addresses. A PSEUDOTRACE sequence comprises 1,000,000 search requests. For each search, we randomly chose a destination from the selected 1000 destination addresses. The data set PSEUDOTRACE100 is similar to PSEUDOTRACE except that only 100 destination addresses were selected to make up the 1,000,000 search requests. Our last data set, PSEUDOTRACE100L16 differs from PSEUDOTRACE100 only in that the 100 destination addresses were chosen so that the length of the longest matching prefix for each is less than 16. So, every search in PSEUDOTRACE100L16 required a search in $OLDP[-1]$. The NONTRACE, PSEUDOTRACE, and PSEUDOTRACE100 data sets represent different degrees of burstiness in the search pattern. In NONTRACE, all search addresses are different. So, this access pattern represents the lowest possible degree of burstiness. In PSEUDOTRACE, since destination addresses that repeat aren't necessarily in consecutive packets, there is some measure of temporal spread among the recurring addresses[3]. PSEUDOTRACE100 has greater burstiness than does PSEUDOTRACE.

For the NONTRACE, PSEUDOTRACE, PSEUDOTRACE100, and PSEUDOTRACE100L16 data sets, the total search time for each data set was measured and then averaged to get the time for a single search. This experiment was repeated 10 times and 10 average times were obtained. The average of these averages is given in Tables 7 through 12. For the PSEUDOTRACE100 and PSEUDOTRACE100L16

[3]By analyzing trace sequences of wide-area traffic networks, we found that the number of different destination addresses in the trace data is 2 to 3 orders of magnitude less than the number of packets. These traces represent a high degree of burstiness. Since there are no publically available traces from a router whose routing table is also available, we simulate real-world searches using the PSEUDOTRACE and PSEUDOTRACE100 search sequences.

data sets, the times are presented only for the base and pure one-level and two-level partitioning schemes. Figures 18 through 21 histogram the average times for Paix. Since the standard deviation in the measured averages was insignificant, we do not report the standard deviations.

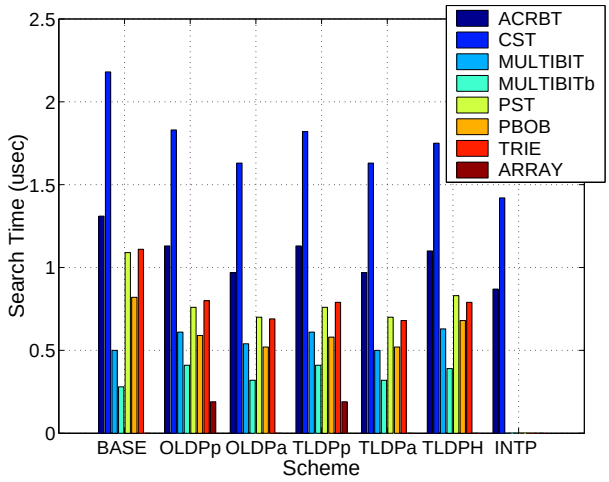

Figure 18: Average NONTRACE search time for Paix

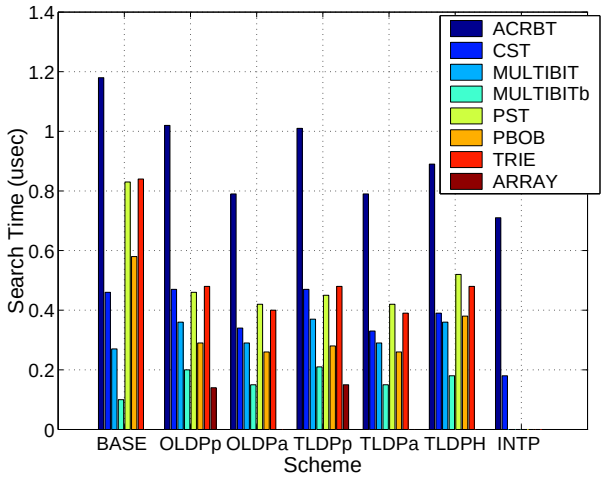

Figure 19: Average PSEUDOTRACE search time for Paix

First, consider measured average search times for only the NONTRACE and PSEUDOTRACE data sets. Notice that the use of the OLDP, TLDP, and INTP schemes reduces the average search time in all cases other than MULTIBIT1p, MULTIBITb1p, MULTIBIT2p, MULTIBITb2p and MULTIBITb2aH, and some of the remaining MULTIBIT cases. For Paix, for example, the MULTIBITb1a search time is

Scheme	Paix	Pb	Aads	MaeWest
ACRBT	1.31	0.98	0.94	0.92
CST	2.18	1.65	1.57	1.54
MULTIBIT	0.50	0.42	0.42	0.41
MULTIBITb	0.28	0.24	0.26	0.25
PST	1.09	0.80	0.75	0.70
PBOB	0.82	0.48	0.45	0.38
TRIE	1.12	0.82	0.75	0.68
ACRBT1p	1.13	0.90	0.87	0.85
CST1p	1.83	1.42	1.35	1.31
MULTIBIT1p	0.61	0.50	0.51	0.51
MULTIBITb1p	0.41	0.35	0.39	0.38
PST1p	0.76	0.59	0.55	0.51
PBOB1p	0.59	0.42	0.39	0.35
TRIE1p	0.80	0.60	0.56	0.51
ARRAY1p	0.19	0.10	0.10	0.09
ACRBT1a	0.97	0.66	0.60	0.57
CST1a	1.63	1.04	0.91	0.85
MULTIBIT1a	0.54	0.36	0.34	0.32
MULTIBITb1a	0.32	0.20	0.22	0.20
PST1a	0.70	0.45	0.39	0.34
PBOB1a	0.52	0.30	0.25	0.20
TRIE1a	0.69	0.41	0.36	0.31
ACRBT2p	1.13	0.89	0.87	0.86
CST2p	1.82	1.41	1.33	1.30
MULTIBIT2p	0.61	0.50	0.52	0.51
MULTIBITb2p	0.41	0.35	0.38	0.37
PST2p	0.76	0.59	0.55	0.50
PBOB2p	0.58	0.41	0.39	0.34
TRIE2p	0.79	0.60	0.56	0.50
ARRAY2p	0.19	0.11	0.10	0.09
ACRBT2a	0.97	0.66	0.60	0.57
CST2a	1.63	1.04	0.90	0.85
MULTIBIT2a	0.50	0.33	0.32	0.30
MULTIBITb2a	0.32	0.21	0.22	0.20
PST2a	0.70	0.45	0.39	0.34
PBOB2a	0.52	0.29	0.25	0.21
TRIE2a	0.68	0.41	0.35	0.30
ACRBTIP	0.87	0.67	0.62	0.61
CSTIP	1.42	1.05	0.96	0.93

Table 7: Average search time (in μsec) for NONTRACE

Scheme	Paix	Pb	Aads	MaeWest
ACRBT2aH	1.10	0.75	0.69	0.65
CST2aH	1.75	1.13	0.99	0.93
MULTIBIT2aH	0.63	0.43	0.42	0.39
MULTIBITb2aH	0.39	0.27	0.28	0.26
PST2aH	0.83	0.57	0.48	0.41
PBOB2aH	0.68	0.41	0.33	0.27
TRIE2aH	0.79	0.55	0.45	0.38

Table 8: Average search time (in μsec) (hash schemes) for NONTRACE

14% larger than that for MULTIBIT on the NONTRACE data set and 50% larger on the PSEUDOTRACE data set. The average search for MULTIBIT2a on the MaeWest database is 27% less than that for MULTIBIT when either the NONTRACE or PSEUDOTRACE data set is used.

The deterioration in performance when partitioning is applied to MULTIBIT and MULTIBITb is to be expected, because partitioning does not reduce the number of cache misses for any search. For example, the height of MULTIBIT is 4 and that of MULTIBITb is 3. So, no search in MULTIBIT results in more than 5 cache misses and in MULTIBITb, no search causes more than 4 cache misses. To search MULTIBIT1p, for example, in the worst case, we must search $OLDP[i]$ (5 cache misses including one to examine the overall root) as well as $OLDP[-1]$ (4 cache misses).

For the Paix database and the NONTRACE data set, PBOB1a and PBOB2a both have a search time that is 37% less than that of PBOB. Although the search time for PBOB2aH is 31% larger than that for PBOB2a, the time is 17% less than that for PBOB. *This finding is important, because it shows the efficacy of the hashing scheme for situations (such as IPv6 with $s = 64$) in which it isn't practical to use an array for the $OLDP$ node.*

Another interesting observation is that the average search time is considerably lower for the PSEUDOTRACE data set than for the NONTRACE data set. This is because of the reduction in average number of cache misses per search when the search sequence is bursty. In fact, increasing the burstiness further using PSEUDOTRACE100 reduces the average search time even further (Table 11 and Figure 20).

For the NONTRACE data set, ARRAY1p and ARRAY2p had the best search time. For the PSEUDOTRACE and PSEUDOTRACE100 data sets, MULTIBITb was fastest and ARRAY1p and ARRAY2p

Scheme	Paix	Pb	Aads	MaeWest
ACRBT	1.18	0.85	0.83	0.81
CST	0.46	0.42	0.42	0.41
MULTIBIT	0.27	0.24	0.23	0.22
MULTIBITb	0.10	0.09	0.10	0.09
PST	0.83	0.60	0.55	0.50
PBOB	0.59	0.34	0.34	0.29
TRIE	0.85	0.63	0.56	0.50
ACRBT1p	1.02	0.79	0.75	0.74
CST1p	0.47	0.40	0.41	0.41
MULTIBIT1p	0.36	0.29	0.34	0.31
MULTIBITb1p	0.20	0.17	0.18	0.19
PST1p	0.46	0.33	0.28	0.28
PBOB1p	0.29	0.19	0.17	0.17
TRIE1p	0.48	0.34	0.33	0.30
ARRAY1p	0.14	0.13	0.11	0.13
ACRBT1a	0.79	0.45	0.40	0.34
CST1a	0.34	0.24	0.21	0.19
MULTIBIT1a	0.29	0.18	0.17	0.16
MULTIBITb1a	0.15	0.12	0.12	0.11
PST1a	0.42	0.23	0.18	0.19
PBOB1a	0.26	0.17	0.16	0.14
TRIE1a	0.40	0.24	0.24	0.21
ACRBT2p	1.01	0.78	0.75	0.74
CST2p	0.47	0.40	0.42	0.42
MULTIBIT2p	0.37	0.31	0.33	0.32
MULTIBITb2p	0.21	0.18	0.19	0.19
PST2p	0.45	0.32	0.28	0.27
PBOB2p	0.28	0.17	0.18	0.16
TRIE2p	0.48	0.31	0.33	0.28
ARRAY2p	0.15	0.12	0.10	0.12
ACRBT2a	0.79	0.47	0.40	0.37
CST2a	0.33	0.24	0.21	0.21
MULTIBIT2a	0.29	0.19	0.17	0.16
MULTIBITb2a	0.15	0.11	0.12	0.11
PST2a	0.42	0.23	0.19	0.19
PBOB2a	0.26	0.16	0.15	0.14
TRIE2a	0.39	0.24	0.22	0.21
ACRBTIP	0.71	0.53	0.49	0.48
CSTIP	0.18	0.17	0.16	0.17

Table 9: Average search time (in μsec) for PSUEDOTRACE

Scheme	Paix	Pb	Aads	MaeWest
ACRBT2aH	0.89	0.51	0.45	0.41
CST2aH	0.39	0.29	0.25	0.24
MULTIBIT2aH	0.36	0.23	0.21	0.20
MULTIBITb2aH	0.18	0.14	0.15	0.14
PST2aH	0.52	0.33	0.26	0.25
PBOB2aH	0.38	0.23	0.20	0.19
TRIE2aH	0.48	0.34	0.28	0.27

Table 10: Average search time (in μsec) (hash schemes) for PSUEDOTRACE

Scheme	Paix	Pb	Aads	MaeWest
ACRBT	0.38	0.31	0.30	0.32
CST	0.22	0.21	0.21	0.25
MULTIBIT	0.14	0.13	0.13	0.13
MULTIBITb	0.07	0.07	0.07	0.07
PST	0.33	0.31	0.29	0.30
PBOB	0.33	0.27	0.27	0.26
TRIE	0.47	0.43	0.42	0.40
ACRBT1p	0.30	0.25	0.24	0.23
CST1p	0.17	0.16	0.16	0.16
MULTIBIT1p	0.16	0.14	0.15	0.14
MULTIBITb1p	0.12	0.11	0.11	0.11
PST1p	0.17	0.15	0.16	0.17
PBOB1p	0.16	0.13	0.13	0.15
TRIE1p	0.27	0.24	0.24	0.24
ARRAY1p	0.12	0.10	0.09	0.11
ACRBT2p	0.30	0.26	0.26	0.25
CST2p	0.18	0.16	0.17	0.16
MULTIBIT2p	0.16	0.15	0.15	0.15
MULTIBITb2p	0.12	0.11	0.11	0.11
PST2p	0.18	0.15	0.17	0.18
PBOB2p	0.16	0.13	0.12	0.15
TRIE2p	0.26	0.24	0.23	0.24
ARRAY2p	0.12	0.11	0.10	0.09

Table 11: Search time (in μsec) for PSUEDOTRACE100

came in next. Although the base ARRAY structure has an $O(n)$ search time complexity, which makes the base ARRAY structure highly non-competitive with the remaining base schemes considered in this paper, the use of partitioning enables the (partitioned) ARRAY scheme to be highly competitive.

Notice that for the NONTRACE, PSEUDOTRACE and PSEUDOTRACE100 data sets, X1p and X2p

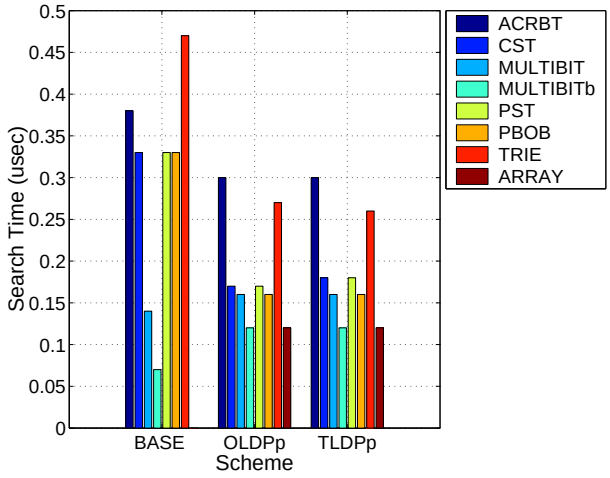

Figure 20: Average PSEUDOTRACE100 search time for Paix

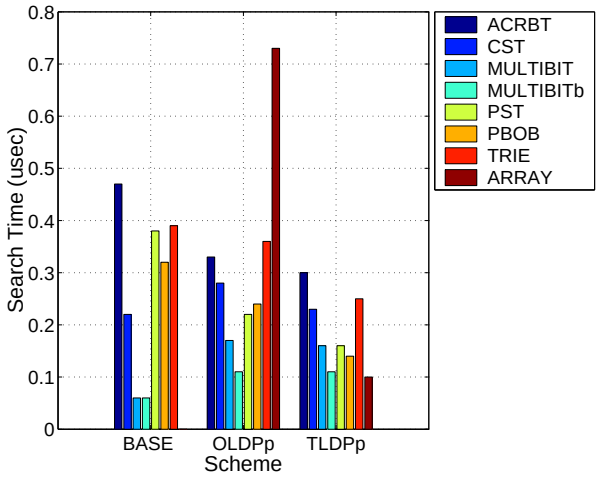

Figure 21: Average PSEUDOTRACE100L16 search time for Paix

have similar average search times. The same is true for X1a and X2a (the PSEUDOTRACE100 times for X1a and X2a are not reported). This result is not surprising since less than 1% of the prefixes have length less than 16. Hence, there is only a small probability that a destination address in NONTRACE and PSEUDOTRACE will require us to examine $OLDP[-1]$. To demonstrate the effectiveness of the TLDP scheme, we use the search sequence PSEUDOTRACE100L16 in which search requires the examination of $OLDP[-1]$. The experimental data of Table 12 and Figure 21 show that the X2p schemes significantly outperform their X1p counterparts. For the Paix database, the average search time for ARRAY2p is

Scheme	Paix	Pb	Aads	MaeWest
ACRBT	0.47	0.40	0.38	0.36
CST	0.22	0.21	0.21	0.26
MULTIBIT	0.06	0.06	0.05	0.06
MULTIBITb	0.06	0.04	0.05	0.05
PST	0.38	0.28	0.28	0.27
PBOB	0.32	0.21	0.25	0.24
TRIE	0.39	0.35	0.34	0.33
ACRBT1p	0.33	0.25	0.26	0.23
CST1p	0.28	0.26	0.28	0.27
MULTIBIT1p	0.17	0.15	0.16	0.16
MULTIBITb1p	0.11	0.11	0.11	0.11
PST1p	0.22	0.18	0.18	0.18
PBOB1p	0.24	0.19	0.19	0.19
TRIE1p	0.36	0.30	0.31	0.29
ARRAY1p	0.73	0.30	0.29	0.38
ACRBT2p	0.30	0.25	0.24	0.23
CST2p	0.23	0.22	0.22	0.21
MULTIBIT2p	0.16	0.14	0.14	0.14
MULTIBITb2p	0.11	0.11	0.10	0.10
PST2p	0.16	0.13	0.13	0.13
PBOB2p	0.14	0.13	0.12	0.12
TRIE2p	0.25	0.21	0.22	0.21
ARRAY2p	0.10	0.08	0.08	0.09

Table 12: Search time (in μsec) for PSUEDOTRACE100L16

14% that for ARRAY1p whereas for PBOB2p, the time is 67% that for PBOB1p. As was the case for the the other search sequences, the run time of MULTIBIT and MULTIBITb are not improved using our partitioning schemes.

Insert Time

To measure the average insert time for each of the data structures, we first obtained a random permutation of the prefixes in each of the databases. Next, the first 75% of the prefixes in this random permutation were inserted into an initially empty data structure. The time to insert the remaining 25% of the prefixes was measured and averaged. This timing experiment was repeated 10 times. Tables 13 and 14 show the average of the 10 average insert times. Figure 22 histograms the average times of Tables 13 and 14 for Paix. Once again, the standard deviation in the average times was insignificant and so isn't reported.

Our insert experiments show that ARRAY1p and ARRAY2p take the least time. When we limit

oursleves to partitioning using base structures whose worst-case performance is better than $O(n)$, we see that the PBOB2a, MULTIBIT2a and MULTIBITb2a structures are competitive and do best for this operation. For example, while an insert in the Paix database takes 19% less time when MULTIBIT2a is used than when a PBOB2a is used, that in the MaeWest takes 15% more time. Generally, the use of OLDP, TLDP, and INTP reduces the insert time. MULTIBIT1p, MULTIBITb1p, MULTIBIT2p, and MULTIBITb2p are exceptions, taking more time for inserts in each of the four databases. MULTIBIT1a, MULTIBITb1a, MULTIBIT2a, and MULTIBITb2a took more time than their base structures on some of the databases and less on others. The insert time for MULTIBIT is about 20% less than that for MULTIBIT1a. On the other hand, the insert time for PBOB2a is between 29% and 39% less than that for PBOB.

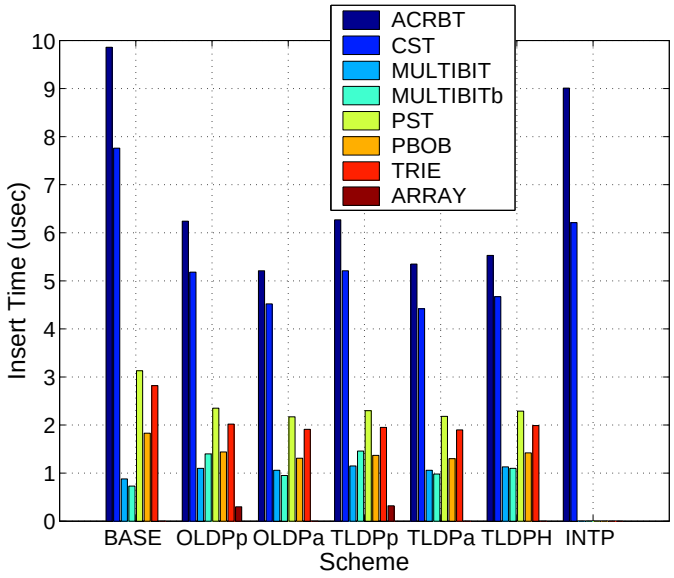

Figure 22: Average prefix insertion time for Paix

Delete Time

To measure the average delete time, we started with the data structure for each database and removed 25% of the prefixes in the database. The prefixes to delete were determined using the permutation generated for the insert time test; the last 25% of these were deleted. Once again, the test was run 10 times and the average of the averages computed. Tables 15 and 16 show the average time to delete a

Scheme	Paix	Pb	Aads	MaeWest
ACRBT	9.86	10.73	10.37	10.20
CST	7.76	6.35	5.95	5.90
MULTIBIT	0.88	0.95	0.97	0.96
MULTIBITb	0.73	1.07	1.22	1.26
PST	3.13	2.60	2.45	2.05
PBOB	1.83	1.54	1.48	1.23
TRIE	2.82	2.31	2.27	2.02
ACRBT1p	6.24	4.96	4.70	4.69
CST1p	5.18	4.16	3.98	4.00
MULTIBIT1p	1.10	1.17	1.24	1.13
MULTIBITb1p	1.40	2.33	2.58	2.52
PST1p	2.35	1.93	1.77	1.52
PBOB1p	1.44	1.17	1.10	0.94
TRIE1p	2.02	1.61	1.51	1.36
ARRAY1p	0.30	0.26	0.28	0.27
ACRBT1a	5.21	3.23	2.90	2.86
CST1a	4.52	2.77	2.40	2.38
MULTIBIT1a	1.06	0.88	0.98	0.75
MULTIBITb1a	0.95	0.91	1.01	0.90
PST1a	2.17	1.67	1.52	1.32
PBOB1a	1.31	0.98	0.91	0.76
TRIE1a	1.91	1.47	1.34	1.18
ACRBT2p	6.27	4.95	4.67	4.69
CST2p	5.21	4.12	3.95	4.00
MULTIBIT2p	1.15	1.26	1.29	1.26
MULTIBITb2p	1.46	2.50	2.64	2.56
PST2p	2.30	1.92	1.76	1.50
PBOB2p	1.37	1.15	1.08	0.93
TRIE2p	1.95	1.60	1.48	1.35
ARRAY2p	0.32	0.24	0.29	0.25
ACRBT2a	5.35	3.28	2.81	2.80
CST2a	4.42	2.73	2.41	2.34
MULTIBIT2a	1.06	0.97	0.96	0.92
MULTIBITb2a	0.98	1.00	1.10	0.98
PST2a	2.18	1.64	1.50	1.29
PBOB2a	1.30	1.00	0.90	0.75
TRIE2a	1.90	1.43	1.34	1.17
ACRBTIP	9.01	7.98	7.53	7.45
CSTIS	6.21	5.19	4.90	4.81

Table 13: Average time to insert a prefix (in μsec)

Scheme	Paix	Pb	Aads	MaeWest
ACRBT2aH	5.53	3.54	3.10	3.01
CST2aH	4.67	2.98	2.60	2.54
MULTIBIT2aH	1.13	1.09	1.04	0.99
MULTIBITb2aH	1.10	1.08	1.17	1.06
PST2aH	2.29	1.75	1.59	1.42
PBOB2aH	1.42	1.08	0.97	0.85
TRIE2aH	1.99	1.53	1.42	1.25

Table 14: Average time to insert a prefix (in μsec) (hash schemes)

prefix over the 10 test runs. Figure 23 histograms the average times of Tables 15 and 16 for Paix. The standard deviations in the average times are not reported as these were insignificant.

As can be seen, the use of OLDP, TLDP, and interval partitioning generally resulted in a reduction in the delete time. The exceptions being MULTIBITb1p and MULTIBITb2p with Paix and Pb. TLDP with array linear lists (i.e., the schemes X2a where X denotes a base scheme such as ACRBT) resulted in the smallest delete times for each of the tested base data structures. The delete time for MULTIBIT2a was between 19% and 62% less than that for MULTIBIT; for PBOB2a, the delete time was between 30% and 39% less than that for PBOB. As was the case for the the search and insert operations, ARRAY1p and ARRAY2p have the least measured average delete time. From among the remaining structures, the delete time is the least for MULTIBIT1a, MULTIBIT2a and PBOB2a. For example, on the Paix database, a delete using MULTIBIT2a takes about 6% less time than when PBOB2a is used; on the MaeWest database, a delete using MULTIBIT2a takes about 12% more time than when PBOB2a is used.

Scheme	Paix	Pb	Aads	MaeWest
ACRBT	9.86	10.73	10.37	10.20
CST	6.34	5.09	4.98	4.84
MULTIBIT	1.34	1.80	2.09	2.06
MULTIBITb	1.46	2.01	2.43	2.44
PST	2.74	2.27	2.12	1.74
PBOB	1.67	1.40	1.31	1.10
TRIE	2.30	1.81	1.75	1.58
ACRBT1p	5.64	4.33	4.03	3.97
CST1p	3.89	3.04	2.98	2.84
MULTIBIT1p	1.25	1.30	1.27	1.37
MULTIBITb1p	1.67	2.03	2.18	2.13
PST1p	2.01	1.69	1.55	1.33
PBOB1p	1.33	1.14	1.06	0.91
TRIE1p	1.67	1.27	1.18	1.11
ARRAY1p	0.30	0.24	0.20	0.22
ACRBT1a	5.10	3.31	2.68	2.71
CST1a	3.57	2.20	2.04	1.97
MULTIBIT1a	1.09	0.88	0.71	0.75
MULTIBITb1a	1.29	1.03	1.04	0.97
PST1a	1.88	1.41	1.27	1.14
PBOB1a	1.21	0.90	0.79	0.70
TRIE1a	1.55	1.14	1.05	0.90
ACRBT2p	5.60	4.20	3.92	3.92
CST2p	3.97	3.00	2.91	2.87
MULTIBIT2p	1.27	1.29	1.29	1.30
MULTIBITb2p	1.70	2.06	2.22	2.16
PST2p	2.00	1.69	1.56	1.32
PBOB2p	1.30	1.13	1.04	0.90
TRIE2p	1.68	1.26	1.18	1.08
ARRAY2p	0.28	0.24	0.21	0.22
ACRBT2a	5.02	3.09	2.75	2.67
CST2a	3.51	2.15	2.03	1.98
MULTIBIT2a	1.10	0.84	0.79	0.76
MULTIBITb2a	1.30	1.00	0.98	0.95
PST2a	1.83	1.40	1.29	1.11
PBOB2a	1.17	0.88	0.80	0.68
TRIE2a	1.57	1.14	1.04	0.86
ACRBTIP	8.13	7.17	6.76	6.65
CSTIP	5.06	4.26	4.16	4.09

Table 15: Average time to delete a prefix (in μsec)

Scheme	Paix	Pb	Aads	MaeWest
ACRBT2aH	5.14	3.18	2.81	2.77
CST2aH	3.67	2.25	2.08	2.03
MULTIBIT2aH	1.18	0.91	0.87	0.84
MULTIBITb2aH	1.35	1.07	1.05	0.97
PST2aH	2.01	1.50	1.37	1.18
PBOB2aH	1.30	0.97	0.97	0.77
TRIE2aH	1.67	1.23	1.13	1.00

Table 16: Average time to delete a prefix (in μsec) (hash schemes)

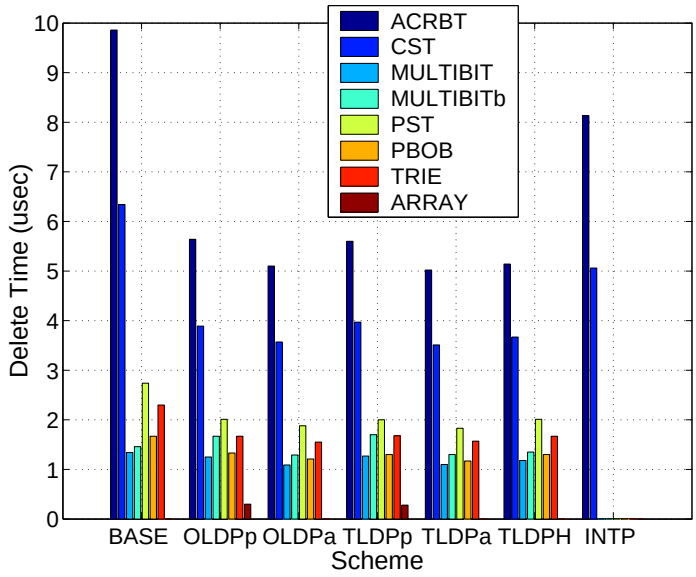

Figure 23: Average deletion time for Paix

6 Conclusion

We have developed two schemes—prefix partitioning and interval partitioning—that may be used to improve the performance of known dynamic router-table structures. The two-level prefix partitioning scheme TLDP also results in memory saving. Although the prefix partitioning schemes were discussed primarily in the context of IPv4 databases, through the use of hashing, the schemes may be effectively used for IPv6 databases. Our experiments with IPv4 databases indicate that the use of hashing degrades the performance of TLDP slightly. However, when s is large (as will be desired for IPv6 databases), the use of an array for the $OLDP$ node is not an option. A similar adaptation of the partition array of [7] to employ a hash table isn't possible.

As indicated by our experiments, our proposed partitioning schemes significantly improve the run time performance of all tested base schemes other than MULTIBIT and MULTIBITb. As an extreme example of this, the performance of the base scheme ARRAY, which is highly impractical for databases of the size used in our test, is enhanced to the point where it handily outperforms the base and partitioned versions of superior base schemes. Although the partitioned array schemes ARRAY1p and ARRAY2p

have unacceptible worst-case performance, our hybrid partitioned schemes that employ array linear lists for small partitions and a structure with good worst-case performance for large partitions provide both a good average and worst-case performance.

In our experiments with the hybrid schemes, we used $\tau = 8$ as the threshold at which we switch from an array linear list to an alternative structure with good worst-case performance. As noted earlier, when $\tau = 0$, we get the corresponding pure scheme and when $\tau = \infty$, we get one of the two partitioned ARRAY schemes ARRAY1p and ARRAY2p. By varying the threshold τ between the two extremes 0 and ∞ the performance of hybrid schemes such as ACRBT1a, MULTIBIT2a, etc. can be varied between that of a pure partitioned scheme and that of ARRAY1p and ARRAY2p. It is anticipated that using $\tau = 128$ (say) would improve the average performance of our hybrid schemes, bringing this average performance close to that of the pure schemes ARRAY1p and ARRAY2p. However, with $\tau = 128$, the worst-case search, insert, and delete times would likely be larger than with $\tau = 8$. The parameter τ may be tuned to get the desired average-case vs. worst-case tradeoff.

The search time for our OLDP and TLDP schemes may be improved by precomputing $bestSmall[i]$ for each nonempty $OLDP[i]$ and each nonempty $TLDP[i]$, $i \geq 0$. For $OLDP[i]$, $bestSmall[i] = OLDP[-1]->lookup(i)$ and for $TLDP[i]$, $bestSmall[i] = TLDP[-1]->lookup(i)$. With $bestSmall$ precomputed in this way, the invocation of $OLDP[-1]->lookup(d)$ in Figure 3 and of $TLDP[-1]->lookup(d)$ in Figure 7 may be eliminated. The major difference between our precomputation of $bestSmall[i]$ and a similar precomputation done for the scheme of Lampson et. al [7] is that we precompute only for nonempty $OLDP[i]$s and $TLDP[i]$s, whereas Lampson et al. [7] precompute for every position of their table. Because of this difference, our precomputation also may be employed in conjunction with our proposed hash table scheme to extend our partitioning schemes to IPv6 databases. Note that although the recommended precomputation reduces the search time, it degrades the update time. For OLDPs, for example, whenever we insert/delete a prefix into $OLDP[-1]$, we would need to update up to 2^s $bestSmall[i]$ values. Of course, the number of $bestSmall[i]$ values that are to be updated also is bounded by the number of *nonempty* $OLDP[i]$s, which, in practice, would be much smaller than 2^s.

References

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

[2] W. Doeringer, G. Karjoth, and M. Nassehi, Routing on longest-matching prefixes, *IEEE/ACM Transactions on Networking*, 4, 1, 1996, 86-97.

[3] F. Ergun, S. Mittra, S. Sahinalp, J. Sharp, and R. Sinha, A dynamic lookup scheme for bursty access patterns, *IEEE INFOCOM*, 2001.

[4] P. Gupta and N. McKeown, Dynamic algorithms with worst-case performance for packet classification, *IFIP Networking*, 2000.

[5] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of Data Structures in C++, W.H. Freeman, NY, 1995, 653 pages.

[6] Merit, Ipma statistics, http://nic.merit.edu/ipma.

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

[8] H. Lu and S. Sahni, $O(\log n)$ dynamic router-tables for prefixes and ranges. Submitted.

[9] H. Lu and S. Sahni, Dynamic IP router-tables using highest-priority matching. Submitted.

[10] H. Lu and S. Sahni, A B-tree dynamic router-table design. Submitted.

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

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

[13] S. Sahni and K. Kim, Efficient construction of fixed-stride multibit tries for IP lookup, *Proceedings 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, 2001.

[14] S. Sahni and K. Kim, Efficient construction of variable-stride multibit tries for IP lookup, *Proceedings IEEE Symposium on Applications and the Internet (SAINT)*, 2002, 220-227.

[15] S. Sahni and K. Kim, $O(\log n)$ dynamic packet routing, *IEEE Symposium on Computers and Communications*, 2002.

[16] S. Sahni and K. Kim, Efficient dynamic lookup for bursty access patterns. Submitted.

[17] S. Sahni, Data structures, algorithms, and applications in Java, McGraw Hill, NY, 2000, 833 pages.

[18] 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.

[19] K. Sklower, A tree-based routing table for Berkeley Unix, Technical Report, University of California, Berkeley, 1993.

[20] V. Srinivasan and G. Varghese, Faster IP lookups using controlled prefix expansion, *ACM Transactions on Computer Systems*, Feb:1-40, 1999.

[21] S. Suri, G. Varghese, and P. Warkhede, Multiway range trees: Scalable IP lookup with fast updates, *GLOBECOM*, 2001.

[22] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, Scalable high speed IP routing lookups, *ACM SIGCOMM*, 1997, 25-36.