Data Structures For IP Lookup With Bursty Access Patterns *

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

Abstract

We propose three router-table data structures for the case when the access pattern is bursty. Longest-prefix matching as well as prefix insertion and deletion take $O(\log n)$ time, amortized time, and expected time in these three structures, respectively. The performance of the proposed structures is compared experimentally using real IPv4 prefix databases.

Keywords: Packet routing, longest matching prefix, dynamic router-tables, bursty access.

1 Introduction

An Internet router table is a set of tuples of the form (p, a), where p is a binary string (or prefix) whose length is at most W (W = 32 for IPv4 destination addresses and W = 128 for IPv6), and a is an output link (or next hop). When a packet with destination address d arrives at a router, we are to find the pair (p, a) in the router table for which $p = LMP(d)$ is a longest matching prefix of d (i.e., p is a prefix of d and there is no longer prefix q of d such that (q, b) is in the table). Once this pair is determined, the packet is sent to ouput link a. The speed at which the router can route packets is limited by the time it takes to perform this table lookup for each packet.

Several software solutions for the IP lookup problem (i.e., finding the longest matching prefix) have been proposed ([1, 2, 3, 4, 5, 8, 10], for example). However, with the exception of [3], none focuses on router-table management for a dynamic environment (i.e., search, insert, and deletes are performed dynamically) in which the access pattern is bursty. In a *bursty* access pattern ([6]) the number of different destination addresses in any subsequence of q packets is $<< q$. That is, if the destination of the current packet is d, there is a high probability that d is also the destination for one or more of the next few packets.

Ergun et al. [3] use ranges to develop a biased skip list structure that performs longest prefix matching in $O(\log n)$ expected time. Their scheme is designed to give good expected performance for "bursty access patterns". However, inserts and deletes take $O(\log n)$ expected time only in the severely restricted and impractical situation when all prefixes in the router table are of the same length. For the more general, and practical, case when the router table comprises prefixes of different length, their scheme takes $O(n)$ expected time for each insert and delete.

Although the CRBT structure of [8] can be used for our proposed environment to perform search, insert, and delete in $O(\log n)$ time each, the CRBT structure does not make explicit use of the assumed bursty nature of the access pattern. An alternative to the CRBT, the ACRBT, is propsed in this paper. This alternative requires more memory than is required by the CRBT. However, experiments indicate that the search, insert, and delete operations are faster in the ACRBT than in the CRBT.

In this paper, we develop also two data structures that explicitly account for the assumed bursty access pattern. The first, biased skip lists with prefix trees (BSLPT), extends the biased skip lists structure proposed in [3] for static router tables. This extension allows us to find longest matching prefixes as well as to insert and delete prefixes in $O(\log n)$ expected time (n is the number of prefixes in the router table). The second data structure, which is a splay-tree data structure called collection of splay trees (CST) allows us to find longest matching prefixes and insert and delete prefixes in $O(\log n)$ amortized time. Experimental results comparing the performance of the proposed structures also are presented.

2 Preliminaries

Each prefix of the router table may be represented as a range $[s, f]$, where s is the start of the range for the prefix and f is the finish of the range for that prefix. For example, when W = 5, the prefix P = 1* matches all destination addresses in the range [10000, 11111] = [16, 31]. So, for prefix P, s = 16 and f = 31. s and f are the end points of the range $[s, f]$.

Let $r_{q+1} = \infty$. Assume that these end points are or-

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

dered so that $r_i < r_{i+1}$, $1 \le i < q$. Each of the intervals $[r_i, r_{i+1}]$, $1 \le i < q$ is called a *basic interval*.

Sahni and Kim [8] propose the use of a collection of red-black trees to determine $LMP(d)$. The CRBT comprises a front-end data structure that is called the *binary interval tree* (BIT) and a back-end data structure called a *collection of prefix trees* (CPT). For any destination address *d*, define the *matching basic interval* to be a basic interval with the property that $r_i \le d \le r_{i+1}$ (note that some *d*s have two matching basic intervals).

The BIT is a binary search tree that is used to search for a matching basic interval for *d*. The BIT comprises internal and external nodes and there is one internal node for each r_i. Since the BIT has q internal nodes, it has $q+1$ external nodes. The first and last of these, in inorder, have no significance. The remaining $q-1$ external nodes, in inorder, represent the $q-1$ basic intervals of the given prefix set. Every external node has three pointers: *startPointer*, *finishPointer*, and *basicIntervalPointer*. For an external node that represents the basic interval $[r_i, r_{i+1}]$, *startPointer* (*finishPointer*) points to the header node of the prefix tree (in the back-end structure) for the prefix (if any) whose range start and finish points are r_i (r_{i+1}). Note that only prefixes whose length is W can have this property. *basicIntervalPointer* points to a prefix node in a prefix tree of the back-end structure.

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. Each prefix tree is a red-black tree. Following parent pointers in the nodes of a prefix tree, we can start at any node and move up to the tree root from where we can determine the prefix represented by this tree.

The search for $LMP(d)$ begins with a search of the BIT for the matching basic interval for *d*. Suppose that external node Q of the BIT represents this matching basic interval. When the destination address equals the left (right) end-point of the matching basic interval and *startPointer* (*finishPointer*) is not null, $LMP(d)$ is pointed to by *startPointer* (*finishPointer*). Otherwise, the back-end CPT is searched for $LMP(d)$. The search of the back-end structure begins at the node $Q.basicIntervalPointer$. By following parent pointers from $Q.basicIntervalPointer$, we reach the header node of the prefix tree that corresponds to $LMP(d)$.

3 Data Structures For Bursty Access

3.1 Biased Skip Lists With Prefix Trees (BSLPT)

Ergun et al. [3] propose a biased skip list (BSL) structure for bursty access. Their structure is suitable for static router tables; that is, router tables in which the prefix set does not change (no prefixes are inserted or deleted). To construct the initial BSL, the n prefixes of the router table are processed to determine the up to $2n-1$ basic intervals they define. The longest matching prefix for destination addresses that fall within each basic interval as well as for destination addresses that equal an end point is determined. A master list of basic intervals along with the determined longest matching prefix information is constructed. This list is indexed into using a skip list structure.

In a biased skip list, ranks are assigned to the basic intervals. $rank(a) < rank(b)$ whenever interval a is accessed more recently than interval b. Basic interval ranks are used to bias the selection of intervals that are represented in skip list chains that are close to the top. Hence, searching for a destination address that is matched by a basic interval of smaller rank takes less time than when the matching interval has higher rank. If the destination of the current packet is d and the matching basic interval for d as determined by the skip list search algorithm is a, then $rank(a)$ becomes 1 and all previous ranks between 1 and $oldRank(a)$ are increased by 1. The skip list structure is updated following each search to reflect the current ranks. Consequently, searching for basic intervals that were recently accessed is faster than searching for those that were last accessed a long time ago. This property makes the BSL structure of [3] perform better on bursty access patterns than on random access patterns. Regardless of the nature of the access, the expected time for a search in a BSL is $O(\log n)$.

The time to insert/delete a prefix when a BSL is used is $O(n)$. By adding the back-end CPT of Sahni and Kim [8] to the BSL of [3], we obatin the BSLPT structure in which search, insert, and delete have expected complexity $O(\log n)$. Notice that a BSL is functionally very similar to the front-end BIT of the CRBT structure of [8]. Both are used to search for the matching basic interval. Unlike the CRBT, which uses a *basicIntervalPointer* to index into a back-end CPT from which $LMP(d)$ may be determined for *d* values that are not matched by prefixes whose length is W, a BSL retains $LMP(d)$ within the master-list node for the matching interval. This is the reason insertion and deletion operations in a BSL take $O(n)$ time. Note that each master-list node of a BSL, like each external node of a BIT, represents a basic interval. In the BSLPT structure, each master-list node has three pointers: *startPointer*, *finishPointer*, and *basicIntervalPointer*. For a master-list node that represents the basic interval $[r_i, r_{i+1}]$, *startPointer* (*finishPointer*) points to the header node of the prefix tree (in the back-end structure) for the prefix (if any) whose range start and finish points are r_i (r_{i+1}). *basicIntervalPointer* points to the prefix node for the basic interval $[r_i, r_{i+1}]$. This prefix node is in a prefix tree of the back-end structure.

To search a BSLPT for $LMP(d)$, we use the BSL search scheme of [3] to search the front-end BSL for a matching basic interval. This search gets us to the master-list node Q for a matching basic interval. When d equals the left (right) end-point of the matching basic interval and *startPointer* (*finishPointer*) is not null, $LMP(d)$ is pointed to by *startPointer* (*finishPointer*). Otherwise, we follow the *basicIntervalPointer* in Q to get into a prefix tree. Then, we follow parent pointers to the header of the prefix tree to determine the prefix that is $LMP(d)$.

To insert a new prefix $P = [s, f]$ (s and f are, respectively, the range start and finish points of P), we note that:

1. When neither s nor f is a new end point, the number of basic intervals is unchanged. Further, since the *basicIntervalPointer* in each master-list node points to the prefix node that represents the same basic interval, none of the *basicIntervalPointer*s in the BSL change (what does change, is the prefix tree these prefix nodes need to be in, but, this change is handled by the insert algorithm for the back-end structure). Only the *startPointer*s and *finishPointer*s in master-list nodes may change when neither s nor f is a new end-point. Further, since these pointers are set only when there is a prefix with $s = f =$ an end point of the basic interval, at most two of these pointers may change when $|P| = W$ ($|P|$ is the length or number of bits in the prefix P). The value of these changed pointers is the header node of the prefix tree for the new prefix P. When $|P| < W$, $s \neq f$ and no pointer in the BSL changes.

2. When s is a new end point, the matching basic interval $[a, b]$ for s splits into two. Note that, because of our assumption that the default prefix * is always present, every destination address d has a matching basic interval. The replacement basic intervals for $i = [a, b]$ are $i1 = [a, s]$ and $i2 = [s, b]$. It is easy to see the following:

   (a) $i1.startPointer = i.startPointer$ and $i2.finishPointer = i.finishPointer$.

   (b) When $|P| = W$, $i1.finishPointer = i2.startPointer =$ pointer to header node of prefix tree for P; $i1.basicIntervalPointer$ and $i2.basicIntervalPointer$ point to prefix nodes for $i1$ and $i2$, respectively. Both of these prefix nodes are in the same prefix tree as was the prefix node for i. This prefix tree can be found by following parent pointers from $i.basicIntervalPointer$.

   (c) Figure 1 shows the situation when $|P| < W$. Since, $|P| < W$, $i1.finishPointer = i2.startPointer =$ **null**. Also, from Figure 1, we see that $i1.basicIntervalPointer = i.basicIntervalPointer$ (note that the prefix node $i.basicIntervalPointer$ now represents the smaller basic interval $i1$) and $i2.basicIntervalPointer$ points to a new prefix node for the interval $i2$. This new prefix node is to be in the prefix tree for P.

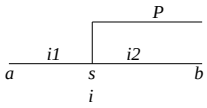


Figure 1: Start point s of P splits $[a, b]$

3. The case when f is a new end point is similar to the preceding case.

4. When both s and f are new end points, both cases (2) and (3) apply.

The algorithm to delete a prefix uses the delete from CPT algorithm of [8] to modify the back-end structure appropriately together with an algorithm to make the necessary changes to the BSL front end. Since these changes to the BSL are the inverse of what happens during an insert, we do not detail them here. When the back-end prefix trees are red-black trees, the expected complexity of a search, insert, and delete is $O(\log n)$.

3.2 Collection Of Splay Trees

Splay trees are binary search trees that self adjust so that the deepest encountered surviving node in any operation becomes the root following the operation. This self-adjusting property makes splay trees a promising data structure for bursty applications. Unfortunately, we cannot simply replace the use of red-black trees in the CRBT structure of [8] with splay trees. This is because, the red-black tree used to represent the BIT of [8] uses two types of nodes: internal and external. Every search has to reach an external node from which it may progress into a back-end prefix tree. Following the splay operation, the reached external node should become the root. This isn't permissible. With this realization, we were faced with the option of either modifying the splay step so that the parent of the reached external node became the root or changing the BIT structure so that the new structure did not have external nodes. The former option was rejected as this would result in a front-end tree whose root always has an external node as one of its children. So, we propose an alternative BIT structure (ABIT) in which we have only internal nodes.

Each (internal) node of the ABIT represents a single basic interval. Therefore, each ABIT node has fields for the

left and right end points of a basic interval, a left and right child pointer, pointers to $LMP(d)$ when d equals either the left or right end point of the basic interval, and a pointer to the corresponding basic-interval node in a prefix tree for the case when d lies between the left and right end points of the basic interval (this is the same as the *basicIntervalPointer* used in CRBT and BSLPT).

The *collection of splay trees* (CST) structure proposed by us for bursty applications of dynamic router tables uses the ABIT structure as the front end. The back end is the same prefix tree structure proposed by us in [8] except that each prefix tree is implemented as a splay tree rather than as a red-black tree. The algorithms to search, insert, and delete for our proposed CST structure are simple adaptations of those for the proposed BSLPT structure. Therefore, further details are not provided. We note that the amortized complexity of these algorithms is $O(\log n)$.

Notice that splay trees also may be used to implement the back-end prefix trees of the BSLPT structure (rather than red-black trees).

4 Experimental Results

We programmed[1] the three bursty-access-schemes described in this paper in C++ and measured the performance of these schemes as well as that of the CRBT scheme of [8]. For the CST structure, we used top-down splay trees when implementing the ABIT and bottom-up splay trees for the back-end prefix trees. Top-down splay trees were used for the ABIT, because past experimental studies show that these perform better than bottom-up splay trees in normal applications. In the case of prefix trees, the search operation begins at the bottom of the splay tree (in normal search applications, this search would begin at the root). So, search time is optimized by using bottom-up rather than top-down splay trees. Our implementation of the BSLPT structure also employed bottom-up splay trees in the back-end structure.

The codes were run on a SUN Ultra Enterprise 4000/5000 computer. The g++ compiler with optimization level -O3 was used. For test data, we used five IPv4 prefix databases obtained from [7], Sept. 13, 2000. The name and number of prefixes in each database are (Paix, 85988), (Pb, 35303), (MaeWest, 30719), (Aads, 27069), and (MaeEast, 22713).

Figure 2 histograms the total memory required by each data structure. The BSLPT takes about twice as much memory as do the remaining structures; the CST is most efficient on memory.

[1]We are grateful to the authors of [3] for providing us their C code for the BSL structure. Our BSLPT code utilized a C++ translation of the BSL code of [3] for the front end and our own code for the back end.

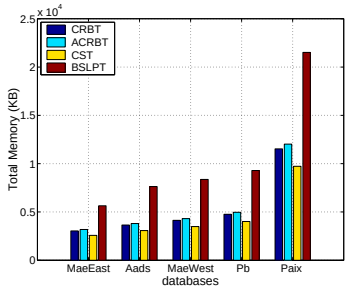
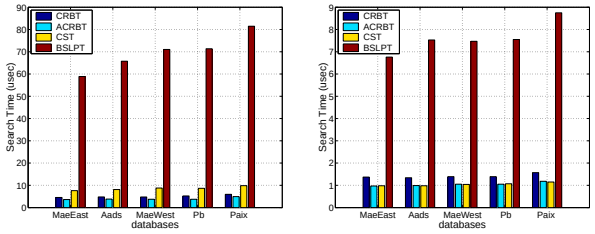


Figure 2: Total memory requirement (in KB)



(a) NODUP (b) DUP10

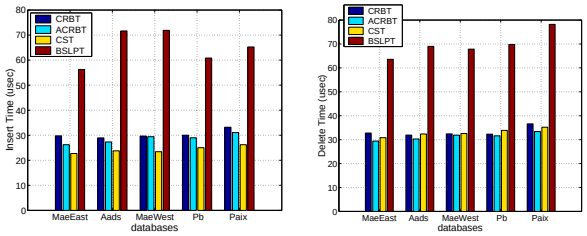
Figure 3: Average search time

Search Time

The destination addresses in the test set NODUP were a random permutation of the end points of the basic intervals corresponding to the database being searched. The data set DUP10 was constructed from NODUP by making 10 consecutive copies of each destination address. In NODUP, all search addresses are different. So, this access pattern represents the lowest possible degree of burstiness. In DUP10, every block of 10 consecutive IP packets has the same destination address, a high degree of burstiness. The average search time is histogrammed in Figures 3(a) and (b), respectively, for NODUP and DUP10.

Searching the database Paix takes about four times as much time per search using the data set NODUP (non-bursty) as it does using the data set DUP10! This is because of the cache effect–the first search for destination d in CRBT potentially causes h cache misses, where h is the height of the BIT. Subsequent searches have (almost) no cache misses because the BIT and back-end nodes needed for the search are still in cache! So, computer caches automatically result in improved performance for bursty access patterns!

The data structures, CST and BSLPT, that are designed to perform better, through a reduction in work (number of comparisons and number of pointers followed), on bursty access patterns show a much greater performance improvement when the access pattern is bursty than when it is not. For example, on the database Paix, both CST and BSLPT



(a) Insert (b) Delete

Figure 4: Average insert and delete times

took about 9 times as much time on the NODUP data set as they did on the DUP10 data set. This, of course, represents the cumulative benefits of the cache effects and of the reduction in work load.

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. Figure 4(a) shows the average of the 10 average insert times.

Our insert experiments show that the BSLPT structure is the clear loser for this operation. The average insert in a BSLPT takes about twice as much time as it does in any of the three tree-based front-end structures. The insert operation is faster in the CST than in the ACRBT, and inserts are faster in the ACRBT than in the CRBT.

Delete Time

To measure the average delete time, we started with the data structure for each database and removed the last 25% of the prefixes in the database. These prefixes were determined using the permutation generated for the insert time test. Figure 4(b) shows the average time to delete a prefix.

As was the case for the search and insert operations, for the delete operation too, the BSLPT structure is the clear loser. A delete in the BSLPT structure takes more than twice the time it takes in any of the tree-based front-end structures. Among the tree-based front-end structures, the delete time is the least for the ACRBT structure.

A more exhaustive experimental comparison of the proposed data structures is given in [9].

5 Conclusion

We have proposed three data structures, ACBRT, BSLPT, and CST, for dynamic router-tables. The last two of these were designed specifically for bursty access patterns. The complexity of longest-prefix matching, insert, and delete is $O(\log n)$ in each of these data structures. In the case of ACBRTs, this is the actual complexity of each operation. However, for BSLPTs, $O(\log n)$ is the expected complexity whereas for CSTs, $O(\log n)$ is the amortized complexity.

Of the structures we tested, the ACRBT is recommended for non-bursty to moderately bursty applications, the CST is recommended for highly bursty applications. Finally, we observe that the add-on data structures, such as a cache list of most-recent destination addresses, suggested in [3] for further improvement in search performance may also be used in conjunction with the data structures of this paper.

References

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

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

[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, B. Prabhakar, and S. Boyd, Near-optimal routing lookups with bounded worst case performance, *IEEE INFOCOM*, 2000.

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

[6] S. Lin and N. McKeown, A simulation study of IP switching, *IEEE INFOCOM*, 2000.

[7] Merit, Ipma statistics, http://nic.merit.edu/ipma, (snapshot on Sep. 13, 2000), 2000.

[8] S. Sahni and K. Kim, $O(\log n)$ dynamic packet routing, 2001, submitted.

[9] S. Sahni and K. Kim, Efficient dynamic lookup for bursty access patterns, University of Florida, 2002.

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