

A Trie-based Algorithm for IP Lookup Problem*

Pinar Altin Yilmaz, Andrey Belenkiy, Necdet Uzun
New Jersey Institute of Technology
Nitin Gogate, Mehmet Toy
Fujitsu Network Communications

Abstract— Increase in routing table sizes, number of updates, traffic, speed of links and migration to IPv6 have made IP address lookup, based on longest prefix matching, a major bottleneck for high performance routers. In this paper, we evaluate and compare several schemes based on complexity analysis and simulation results. We present a trie based scheme, which we call Linked List Cascade Addressable Trie (LLCAT), which is easy to be implemented in hardware and allows routing table update operations to be performed incrementally requiring very few memory operations guaranteed for worst case to satisfy requirements of dynamic routing tables in high speed routers. We also consider compression schemes to be applied to our algorithm to improve memory consumption and search time.

I. INTRODUCTION

The number of Internet hosts shows an exponential growth [7]. As new multimedia applications are introduced, and conveniences of Internet are discovered by more and more people, the number of users and the amount of traffic are also increasing. As higher bandwidth is offered by optical networks, router performance becomes the key.

For each arriving packet, routers perform longest matching prefix lookup based on the destination address to find the corresponding output interface.

The prefix length distribution of a routing table is not uniform, showing a high concentration at 16 and 24 bit prefixes as in Fig. 1, hitting maximum at 24 bits with more than 26,000 entries in this routing table of more than 47,000 entries.

The whole idea of prefixes is to aggregate the addresses so that the routing table size is kept small. More customers are choosing multiple network service providers for redundancy, load sharing and flexibility, their networks are "multi-homed". More than 25 percent of prefixes are multi-homed and non-aggregatable and showing a steep linear rate of growth [8]. As studied in [8] and [11], degradation in Internet hierarchy and the resulting increase in the globally visible paths will result in much larger routing tables and a potentially higher routing instability. Routing update information tends to be extremely bursty. Core Internet routers receive bursts of updates at rates exceeding 100 prefix announcements per second. Most Internet outages are short-lived, lasting in the order of seconds or minutes. Informal experiments suggest that sufficiently high rates of

pathological updates (300 updates per second) are enough to crash several routers [8].

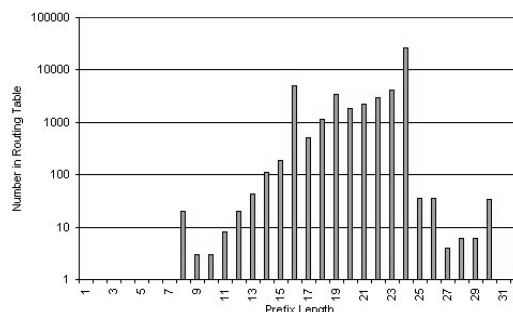


Fig. 1: Histogram of prefixes from Mac-East NAP on September 2, 1999. (logarithmic scale) [11]

Section II will present a summary of some of the available schemes for routing lookup. Section III will introduce LLCAT algorithm and discuss its performance in the light of simulation results. Section IV will present the conclusions.

II. CLASSIFICATION OF ROUTING LOOKUP ALGORITHMS

Most solutions proposed and implemented are based on tries. Most popular and widely implemented is perhaps Patricia Trie [14]. Patricia trie is a binary radix trie with one-way branching removed, also referred to as a path compressed binary trie because it eliminates the traversal of the full path to a leaf node if there is only one leaf in that branch by introducing a skip factor, which is the length of the path to be overlooked. It suffers from backtracking which occurs when a match is not found at a leaf node. Worst-case complexity is $O(W^2)$ where W is number of bits in the address.

To simplify deletion and avoid recursive backtracking in Patricia tries, *dynamic prefix tries* [4] have been introduced. At each node, a link to its parent, left child and right child the routing decisions for left and right branches are stored with index key, which determines how many bits are common in the prefixes inserted in the branch. The look-up times have an upper bound of $2 \times W$ iterations. Insertion and deletion operations do not depend directly on the size of the trie but linearly on the height of the trie.

Level-Compressed Tries [12,13] is based on the idea to compress parts of a trie that are densely populated. The worst case look-up is bounded by the maximal path through the trie. The average depth of the trie grows very slowly. Multicast support is available by treating multicast addresses

* This work is partially sponsored by Fujitsu Network Communications.

as 32-bit prefixes using a bit flag. LC-trie is built periodically from a base vector, which has to be kept sorted, or sorted periodically.

Expanded Tries [15] reduce a set of arbitrary length prefixes to a predefined set of prefixes by prefix expansion. Sub-prefixes of the same length are placed in each multi-bit node of the trie. Shorter prefixes will be expanded into many entries by padding bits. The resulting trie may have equal length of prefixes, e.g. 8-bit sub-prefixes at each level, or different length of sub-prefixes at each level, e.g. 16,8 and 8 bits at three levels. Such tries are *fixed-stride*. In each level of the trie, there may be nodes carrying prefixes of different length, e.g. at the same level, say 2, there may be a node carrying 8-bit prefixes and another carrying 4-bit prefixes, such tries are *variable-stride*. To improve locality and storage requirements, leaf-pushing and packed arrays can be used. When the prefixes are expanded, the prefix length information which is necessary for updates is lost. The solution is to keep the original prefix table intact, update this table and generate the trie from scratch when there are changes in the table as described by the authors in US Patent #6,011,795. Another solution suggested by the authors is to maintain original unexpanded prefix information separately. One-bit trie for each node is used to store actual prefix data before expansion. Fixed strides are desirable because of their simplicity, fast optimization times, and faster search times. Search time is linear on the number of levels of the trie. Updates depend linearly on length of the prefix and maximum size of the trie node.

Multiway and Multicolumn search / 6-way search [9] treats each prefix as a range and encodes using start and end of range. The longest match lookup becomes finding the narrowest enclosing range. Entries are arranged in a binary search table and a mapping between consecutive regions in the binary search table and corresponding prefixes is precomputed. Worst case lookup takes 10 memory accesses for 38,000 entries. Locality inherent in processor cache is exploited for multiway search. Incremental insertion is possible but building a table from scratch achieves better results in the order of $O(N)$ where there are N prefixes in the table.

Lulea Scheme [3] uses a trie with fixed levels of 16,8, and 8 bits. The result of a search on a level is either an index into next-hop table or an index into an array of chunks for the next level. Large trie nodes can be compressed by counting the number of bits set in a large bitmap. Leaf pushing is used to complete the trie at each level, thus incremental insertion and deletion will be slow, and forwarding table is built during a single pass over all routing entities. Table size and building time are linear in the number of routing table entries.

Multi-Resolution Tries [16] is also an expanded trie with fixed strides. Each node of the trie contain forwarding information in its entries. Actual prefix information (length and decision) is stored as a linked list per entry. These linked lists can be sorted or organized into a trie for faster access.

Garbage collection is necessary to remove nodes with no decisions and no children. The authors have implemented the trie with a small strides with 12,4,4,2,2,2,2,1,1,1,1 bits, which generates a worst case lookup of 11 memory accesses, on average 4 memory accesses.

Solutions based on caching [2] and CAMs (Content Addressable Memory) [6,10] are also available. A mixture of hashing and trie mechanisms is *Binary Search on Levels* [18].

III. LLCAT: LINKED LIST CASCADE ADDRESSABLE TRIE

LLCAT is an expanded trie with fixed strides, which makes memory management very simple [1]. The basic idea of expanded tries has been described in [15]. However LLCAT algorithm differs from expanded tries presented in the literature in the fact that it has a new approach for memory management and update operations where it is necessary to maintain original prefix data which is lost during expansion. The trie nodes are represented as segments in memory and each entry in the node is a word for hardware implementation. Destination address and prefixes are divided into blocks of K bits. K may be 4 or 8. For IPv6, a larger K may be appropriate. The search is bounded by the number of levels in the trie determined by the value of K . If $K=8$, then it takes 4 memory accesses in the worst case. The improvement of LLCAT over other expanded tries is that it provides a more efficient approach for updates. Instead of keeping tries or linked lists to hold original prefix information, it keeps original prefix decisions and a bitmap of inserted prefixes in each entry. Update operations depend on number of levels, size of the trie node, and distribution of the prefixes, not necessarily on the size of the routing table.

A. Algorithm Description

The algorithm described here is designed to be easily realized in hardware. It does not contain any complex computational functions, only data transfer and bit-wise logical operations. It may also be used for label-to-flow mapping, address filtering, etc. Physical memory is divided into segments, which can be uniquely addressed having equal number of words. Each word in a segment is identified by an offset.

The search string, destination address (DA) is divided into K equal substrings. Each of them will become an offset during the search. Every node will contain 2^K words in order to accommodate every possible combination of a K -bit substring as in Fig. 2. Each word contains the following fields:

- *Stop bit (S)* indicates whether the search stops here or will follow a link to the next level of the trie.
- *Valid bit (V)* indicates whether the route information located in this word is valid or not.
- *Route for a full-prefix flag (F)* indicates when the R field contains routing information for the full prefix (when 1) or for a sub-prefix (when 0).

- *Life Children Counter (CC)* shows the number of words in the node that carry useful information. In this context, useful information is either a word with a valid route ($V=1$) or a word with a valid *FP* ($S=0$). The first entry of a segment contains *CC*.

- *Output Port Address or Route (R)* contains the routing decision for the prefix represented by the entry.

- *Valid sub-prefix pattern (VSP)* contains the mask of possible sub-prefix patterns with less than K bit length for this word. In Fig. 2, offset 1111 on level 2, has the *VSP* field set to 101. This means that sub-prefixes 1 and 111 are inserted in this entry.

- *Forward Pointer (FP)* contains a pointer to a segment in the next level of the trie.

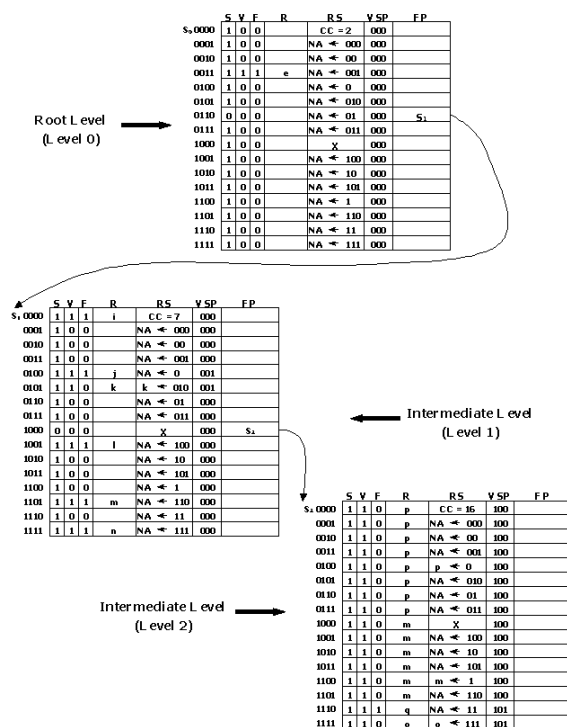


Fig. 2: Variable prefix enhancement

The address of the first node, which is the root of the trie, is always fixed at 0. The first substring of DA gives the offset of the first entry to be visited. The address of the node of the next level is given in the *FP* field, and the address of the entry to be visited next is obtained by concatenating the content of *FP* with the next substring of DA. When the search reads the word in memory, it checks all the flags. If $V=1$, then *R* will be stored in a temporary register, so that at any point, the most recent routing decision encountered is available. The process continues until $S=1$. At this point, the search knows that it has to stop, and the *R* field at this location is used to forward this packet if $V=1$. If $V=0$, the last valid *R* stored in the temporary register is used as the forwarding address.

Refer to Fig. 2 for the following discussion. Number of sub-prefixes that must be stored in a node is 2^{K-1} . In the case where $K=4$, the number of words in the node is 16, and there are 14 possible sub-prefixes: 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, and 111. Original routing decisions for a sub-prefix can be stored in entries whose offset is determined by the sub-prefix itself. To determine the offset of a sub-prefix, the following rule applies: 3-bit sub-prefix & 1, 2-bit sub-prefix & 10, 1-bit sub-prefix & 100, where & stands for concatenation, constituting the offset of the word in the node where the route for the sub-prefix is to be found. As a general rule, the sub-prefix is concatenated with 1 if it is $K-1$ bit long, with 10 if it is $K-2$ bit long, with 100 if it is $K-3$ bit long, and so on. The unused locations of the *CC* field can be used for this purpose. For example, in Fig. 2, to insert sub-prefix "0110 010• k" in the trie the following procedure is used. On the root level, first four bits give the entry to be visited and a pointer to the next level is found. On level 1, only 3 bits remain. This means that a sub-prefix insertion is performed. Applying the address rule for 3-bit sub-prefix, it is concatenated with 1 to obtain 0101, so that the *RS* field of the entry gets the value k.

Insertion of the sub-prefix is carried out by traversing down the trie and creating new nodes along the way if $S=1$, until the last node is reached. For sub-prefix insertion, the offset of the entry is determined from the address rule by concatenating appropriate bits to the sub-prefix and routing information is written to the *RS* field. Then, it is necessary to expand the sub-prefix into all possible full prefixes. For example sub-prefix 01 is expanded into 0100, 0101, 0110, and 0111. Notice that after expansion, offsets are sequential. In terms of hardware, it may be implemented simply by concatenating values of the counter to the sub-prefix. Because the sub-prefix has a length of 2 bits, the second bit in *VSP* fields is affected. *VSP* in every affected word will be OR'ed with 010 (second bit asserted). If $V=0$, then routing information associated with the inserted sub-prefix is stored in *R*, *V* flag is asserted to 1, and *CC* will be incremented, depending on *S*. If $V=1$, then this word has been accounted for already and *CC* should remain unchanged but *R* must be replaced if the new decision is for a longer prefix, that is if $F=0$ and no higher bits in *VSP* are asserted.

Deletion of a sub-prefix works in the following way. The sub-prefix is expanded to its full prefixes and all of the entries addressed by these prefixes are examined. First, the *VSP* pattern is going to be changed. Corresponding bit is set to 0 by performing a logical AND, e.g. if we try to remove the sub-prefix 10, then all the *VSP* fields in effected entries are AND'ed with 101 to set the second bit to 0. If $F=1$, the process moves on to the next word. If $F=0$ and no higher-order bits in *VSP*, corresponding to longer sub-prefixes, are asserted, the deletion procedure has to determine longest sub-prefix and look up *RS* field of the corresponding entry. If no bits of the *VSP* are 1, the deletion operation has to invalidate

R , by changing V to 0 and decrementing CC of this node. Only then deletion moves on to the next word.

To keep track of which segments are used in the trie, a list of idle nodes (ILL) is maintained. Each node in ILL has its FP set to the address of next node in the list. Pointers to head and tail of the list are stored separately. Nodes to be inserted are extracted from the head, and deleted nodes are added to the tail.

There are several performance enhancements that can be applied to the trie. First, it can be observed that there are no prefixes shorter than 8 bits. Therefore, for IPv4, it is beneficial to combine levels 0 and 1 and just keep it in the small fast cache. This enhancement saves one clock cycle. Thus, the root node will have $2^8=256$ words, and the first stride is 8 bits. Also, it may be observed that VSP and F fields may be combined into one single field of K bits, where each bit will function as a flag if there is routing information. If $VSP=0$ then there is no routing information present. For an entry to be valid, there is either routing information present and/or a forward pointer exists as signaled by the field S . Therefore, we do not require the V field to check the validity of an entry.

B. Performance analysis

In this section, the worst and best case timing and memory requirements are presented. The worst-case requirements for timing are defined by the time necessary to look up or update a 29-bit prefix for IPv4 and a 121-bit prefix for IPv6. The worst case for memory requirements occurs when all the entries are located in separate nodes of the trie. Each node has only one valid child organizing the nodes as a number of linked lists.

Number of memory operations required for search is bounded by the depth of the trie, which are 7 for $K=4$, and 4 for $K=8$ considering IPv4. For $K=4$, worst case requires exactly 27 memory operations for insertion and exactly 46 memory operations for deletion [19].

The speed of the memory is a main factor affecting the speed of the lookup. Cost-efficient design can use slower memory. For IPv4, using 10 nanoseconds (nsec) of static random access memory (SRAM) requires only 70 nsec for a search. For example, on an OC-48 interface of a IPv4 router, the smallest packet with size of 64 bytes corresponds to 205 nsec. Therefore, after the search is done, there are 135 nsec for other tasks, such as insertion or deletion of an entry and collection of statistical data. Deletion of an entry requires at most 46 memory cycles, or 460 nsec. Therefore, one insertion or deletion can be performed for every four lookups. For IPv6, the same hardware allows one trie update per 80 lookups. This means that it is possible to perform more than 30,000 updates per second, which is more than enough to satisfy the most demanding routing algorithms. These numbers are guaranteed for the worst case with respect to the size of incoming packets and the location of the entries in the trie.

C. Simulation of LLCAT

We have implemented LLCAT algorithm in C and tested with data collected from several Internet sources [5,11,17]. The implementation is not a software solution, rather a simulation of the hardware design of LLCAT. Therefore, time for operations is not relevant, however, number of memory operations is relevant.

We have five routing tables. Aads, Mae-East, Mae-West are the main US exchange points provided by [11]. Vbns table is from the vBNS experimental network [17]. Funet table is obtained from [5], which comes from a European ISP and dates back to 1997. Memory consumption of the algorithm for the trie structure is given in Table 1 for different tables. Fig. 3 gives the memory consumption for each of the five tables according to number of prefixes in the table. The drop at Funet table memory size at 41709 prefixes indicates that prefix distribution is more effective on memory consumption than the number of prefixes in the table.

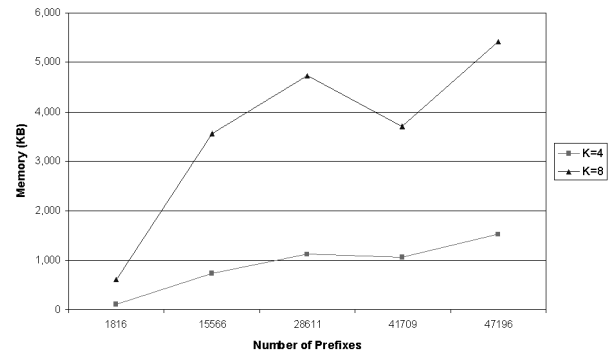


Fig. 3 : Memory consumption of LLCAT with respect to number of prefixes.

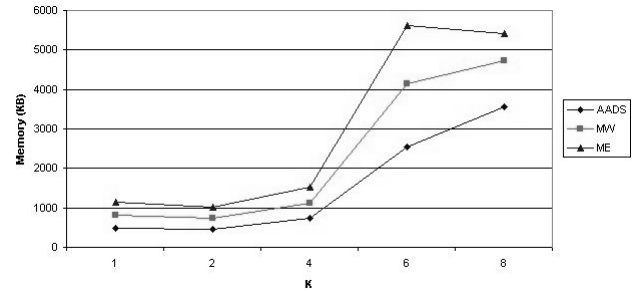


Fig. 4 : Memory consumption of LLCAT with respect to different values of K .

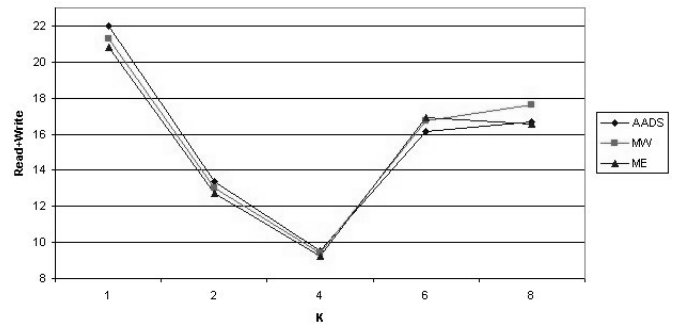


Fig. 5 : Memory operation counts during table build-up of LLCAT.

Router	AADS		MAE-WEST		MAE-EAST		FUNET		VBNS	
Number of Prefixes	15566		28611		47196		41709		1816	
Average Prefix Length	22.05		21.88		22.04		22.02		21.01	
K	4	8	4	8	4	8	4	8	4	8
Number of Words	139,312	647,424	212,608	860,928	282,384	987,136	202,736	674,304	23,968	117,504
Memory Size (KB)	731	3,556	1,116	4,729	1,517	5,423	1,064	3,704	117	602

Table 1: Routing tables used in simulations.

Router	AADS					MAE-WEST					MAE-EAST				
K	1	2	4	6	8	1	2	4	6	8	1	2	4	6	8
Number of Nodes	48,038	21,994	8,691	7,400	2,528	76,656	34,813	13,272	11,777	3,362	108,278	48,219	17,633	16,011	3,855
Memory Size (KB)	494	452	731	2,545	3,556	806	732	1,116	4,142	4,729	1,138	1,014	1,517	5,630	5,423
Avg. Read+Write	21.98	13.38	9.56	16.15	16.68	21.26	13.01	9.44	16.74	17.64	20.84	12.73	9.26	16.89	16.58
Max. Read+Write	44	27	22	69	260	48	30	25	69	261	45	27	23	69	261
Avg. Read per Lookup with NJIT Trace	10.78	6.20	3.85	3.17	2.60	11.56	6.55	4.02	3.27	2.69	11.72	6.62	4.04	3.30	2.70

Table 2: Memory operation counts during table build-up (rows 6-11) and lookup with real packet data (last row).

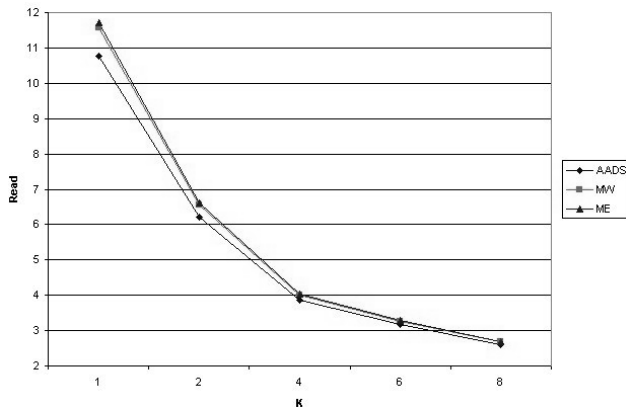


Fig. 6 : Memory Read Operation during Lookup with Real Packet Traces from NJIT network.

Router	AADS		MAE-WEST		MAE-EAST	
K	4	8	4	8	4	8
Avg.Read	5.96	16.50	5.59	10.83	5.57	11.94
Avg.Write	2.51	14.69	2.05	9.03	2.03	10.03
Max Read	12	130	13	131	13	131
Max Write	12	128	11	131	11	131
Avg.Op	8.48	31.20	7.64	19.86	7.60	22.03
Max Op.	20	258	22	259	22	259

Table 3: Memory operation counts for incremental insertion for LLCAT.

Router	AADS		MAE-WEST		MAE-EAST	
K	4	8	4	8	4	8
Avg.Read	6.80	8.72	6.85	8.64	6.71	9.17
Avg.Write	3.64	6.82	3.53	6.23	3.31	6.67
Max Read	15	68	15	71	14	71
Max Write	16	68	14	65	12	65
Avg.Op	10.45	15.54	10.38	14.87	10.03	15.84
Max Op.	31	136	29	136	26	136

Table 4: Memory operation counts for incremental deletion for LLCAT.

We used packet traces obtained from New Jersey Institute of Technology network, and used it with Aads, Mae-East and Mae-West tables with different values for K . The results are summarized in Table 2 showing average and maximum number of memory operations required for each prefix insertion during the build-up of the table, and in the last row,

average number of read operations per look-up for the packets in trace. Trade-off between memory consumption and required number of accesses which determines the speed of algorithm is very apparent as can be seen in Fig. 4, 5, and 6, which summarize results of Table 2. Notice that $K=1$ is indeed similar to a binary tree which provides one way branching, either left or right at each node depending of the value of the bit. It appears that $K=4$ provides an almost optimal point for memory and speed in the case of IPv4. Since most prefixes are 16 or 24 bits, it is important that K should be a divisor of 16 and 24 so that the words in trie nodes are better utilized.

For updates, we have collected the next-day's routing table and extracted the differences and fed them in random order as update requests. Table 3 and 4 gives the memory operation counts for update operations.

We have observed that average case behavior of LLCAT, both in terms of memory operation counts and memory consumption turned out to be quite promising. It results that $K=4$ is suitable for IPv4, and $K=8$ may prove to be useful for IPv8.

We have also considered path compression as implemented in Patricia trie. For this, we have examined the trie structure for the existence of chains of empty nodes. Empty nodes are nodes which do not contain any routing decisions in the entries and each empty node has only one forward pointer and therefore children count (CC) is 1. These intermediary nodes can be eliminated by introducing a pointer from the beginning of the chain to the last node of the chain, thus decreasing the search time and memory consumption. We have seen that for values of $K \geq 4$ less than 6 per cent of the trie nodes were involved in chains, which makes the effort of skipping nodes unnecessary, simply because prefixes are already packed up in big nodes. For $K=1$ almost half of the nodes and for $K=2$ a quarter of the nodes were involved in chains, which may improve average look-up time at the expense of more complex search and update algorithms. Playing with the value of K appears to be a more

sensible approach, which will definitely improve look-up time with a small increase in memory but without increasing the complexity of the algorithm. As a result of our simulations with IPv4 data we believe that $K=4$ provides an almost optimal compromise between memory consumption and number of memory accesses required for insertions, deletion and lookup without altering the elegance and simplicity of the algorithm.

Algorithm	24 bit prefix lookup (nsec)	Worst case lookup (nsec)	Memory for 40,000 prefixes (KB)
Patricia Trie	1500	2500	3262
6-way search	490	490	950
Binary Search on levels	250	650	1600
Lulea Scheme	349	409	160
LC Trie	1000	n/a	700
Expanded Trie	206	n/a	450
LLCAT	$5 \times 15 = 75$	$7 \times 15 = 105$	1060

Table 5: Lookup times for various schemes on a 300 Mhz Pentium II with 15 nsec 512 KB L2 cache.

Algorithm	Insertion	Deletion	Lookup
Patricia Trie	n/a	n/a	$O(W^2)$
Dynamic Prefix Trie	$O(N)$	$O(N)$	$O(\log_i(N)+1)$
6-way search	$O(N)$	$O(N)$	$O(\log_i(N)+W)$
Binary Search	n/a	n/a	$O(W/s)$
Lulea Scheme	n/a	n/a	$O(\log W)$
Binary Search on Levels	n/a	n/a	$O(W/s)$
Large Memory Architecture	n/a	n/a	$O(W/s)$
LC-Trie	n/a	n/a	$O(W/s)$
Expanded Tries	n/a	n/a	$O(W/s)$
LLCAT	$O(W/K+2^{K-1})$	$O(W/K+2^{K-1})$	$O(W/K)$

Table 6: Worst-case complexity of various algorithms (s and K are constants depending on algorithm).

I. CONCLUSION

For evaluating routing lookup algorithms, there are several criteria. Many schemes have been proposed especially in the last two years foreseeing the need for faster routing in Gigabit/Terabit networks, in which router performance becomes a major bottleneck. The relevant weights to each criteria are assigned uniquely for different environments and applications. Each scheme has its own strengths and weaknesses compared in Table 5 and 6.

We have proposed a trie based algorithm which we call LLCAT, with particular strength in update operations and ease of implementation in hardware. Simulation results suggest that LLCAT will meet the performance demands of high speed line-rate lookup and heavy loads of update requests. We believe that such demands may be only met by sophisticated hardware solutions. Demands will increase with the introduction of wireless networks with mobile subnets. Routing lookup with frequent updates will continue to be a bottleneck for high performance routers.

REFERENCES

- [1] A. Belenkiy and N. Uzun, "Deterministic IP Table Look-Up at Wire Speed," INET'99: The Internet Global Summit, San Jose, CA, June 1999.
- [2] Tzi-Cker Chiueh, Prashant Pradhan, "Cache Memory Design for Internet Processors," Proceedings of IEEE Hot Interconnects-VII, August 18-20, 1999.
- [3] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," presented at the SIGCOMM'97, Cannes, France.
- [4] W. Doeringer, G. Karjoth, and M. Nassehi, "Routing on longest-matching prefixes," IEEE/ACM Trans. Networking, vol. 4, pp. 86-97, Feb. 1996.
- [5] Funet Routing Table and Traces, <http://www.nada.kth.se/~snilsson/public/code/router/>
- [6] P. Gupta, N. McKeown, and S. Lin, "Routing lookups in hardware at memory access speeds," in Proc. IEEE INFOCOM'98 Conf., pp. 1240-1247.
- [7] Internet Software Consortium, "Internet Domain Survey, July 1999" <http://www.isc.org/dsview.cgi?domainsurvey/WWW-9907/report.html>
- [8] C. Labovitz, G. Malan, and F. Jahanian, "Internet routing instability," IEEE Trans. Networking, Vol. 6 Number 5, pp. 515, Oct. 1998.
- [9] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," IEEE Trans. Networking, Vol. 7 Number 3, June 1999, pp. 324.
- [10] A. McAuley and P. Francis, "Fast routing table lookup using CAMs," INFOCOM'93, p. 1382-1391, March-April 1993.
- [11] Merit Network, Inc., "Internet Performance Measurement and Analysis Project" <http://www.merit.edu/IPMA>
- [12] S. Nilsson and G. Karlsson, "Fast address look-up for Internet routers," in Proc. IEEE Broadband Communications'98, Stuttgart, Germany.
- [13] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries", IEEE Journal on Selected Areas in Communications, Volume 17 Number 6, June 1999, pp. 1083.
- [14] K. Sklower, "A tree-based routing table for Berkeley unix," presented at the 1991 Winter Usenix Conf., Dallas, TX.
- [15] V. Srinivasan and G. Varghese, "Fast IP lookups using controlled prefix expansion," ACM TOCS, vol. 17, pp. 1-40, Feb. 1999.
- [16] H.-Y. Tzeng, T. Przygienda, "On Fast Address-Lookup Algorithms", IEEE Journal on Selected Areas in Communications, Volume 17 Number 6, June 1999, pp. 1067.
- [17] "VBNS Route Information", <http://www.vbns.net/route/Allsnap.rt.html>
- [18] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in Proc. ACM SIGCOMM'97 Conf., Cannes, France, pp. 25-35.
- [19] P. Yilmaz, "An Algorithm for Fast IP Route Lookup and Update," Master's Thesis, New Jersey Institute of Technology, May 2000.