Performance Enhancement of IP Forwarding by Reducing Routing Table Construction Time

Pi-Chung Wang, Chia-Tai Chan, and Yaw-Chung Chen

***Abstract*—In previous work, Lampson *et al.* proposed an IP lookup algorithm which performs binary search on prefixes (BSP) [3]. The algorithm is attractive for IPv6 because of its bounded worst-case memory requirement. Although for the sake of fast forwarding, the cost paid for the slowing down insertion is reasonable, the performance of routing-table reconstruction in BGP is too time-consuming to handle the frequent route updates. In this letter, we propose a fast forwarding-table construction algorithm which can handle more than 3600 route updates per second. Moreover, it is simple enough to fulfill the need of fast packet forwarding.**

***Index Terms*—Gigabit networking, Internet, IP address lookup.**

I. Introduction

THERE has been a remarkable interest in the organization of routing tables during the past few years. The proposals include both hardware and software solutions [1]–[3], [6]. In [3], the proposed IP lookup algorithm pre-computes both the prefix corresponding to each region and that corresponding to each exact match. By using additional pre-computation, it can perform a prefix match through a binary search in a sorted array. With the cache line alignment, it can achieve more than 2 MPPS (million of packets per second) worst-case performance using 200-MHz CPU. This scheme is attractive for IPv6 because of its bounded memory requirement. Although for the sake of fast forwarding, its cost for the slowing down insertion is reasonable, the performance of routing-table reconstruction is too time-consuming to handle the frequent route updates. For example, in the worst case, the table construction may take 5.8 s. For prefixes with length smaller than 16 bits, the worst-case update time is about 1.25 ms, while the estimated worst and average update time for prefixes with length larger than 16 bits are 352 and 20 ms, respectively [3]. Obviously, the routing table construction is not fast enough to handle the rapid route update (i.e., 100 route updates per second) in a backbone router with BGP implementation [7].

In this work, we aim at enhancing the BSP for prefixes longer than 16 bits using a fast forwarding-table construction algorithm. In BSP, the forwarding-table construction consists of two phases: the sorting and the stack operation. By embedding the sorting into the stack operation, the proposed algorithm can finish the table construction much faster than that of BSP. In addition, we can solve the duplicate-address entry problem easily by solely processing the sorted prefixes. We evaluate the performance of our proposed algorithm based on routing-table update rate. Experimental results show that it handles more than 3600 route-updates per second, this indicates that our proposed algorithm outperforms the BSP. The rest of this article is organized as follows. Section II presents the proposed algorithm. The performance evaluation is addressed in Section III. Section IV concludes the work.

Manuscript received September 6, 2000. The associate editor coordinating the review of this letter and approving it for publication was Dr. I. Venieris.

P.-C. Wang and Y.-C. Chen are with the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C. (e-mail: {pcwang@csie.nctu.edu.tw; ycchen@csie.nctu.edu.tw).

C.-T. Chan was with the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C. He is now with the Telecommunication Laboratories, Chunghwa Telecom Company, Ltd., Taipei, Taiwan, R.O.C. (e-mail: ctchan@cht.com.tw).

Publisher Item Identifier S 1089-7798(01)04501-X.

II. Proposed IP Forwarding Table Construction Algorithm

The table construction of the BSP consists of two parts: the pre-computation for the searchable entries and the multiway search tree construction. One entry with starting address (padded with 0's) and another with ending address (padded with 1s) for each route prefix are generated. In addition, two fields are attached: "**equal**" and "**large**." The former points to the best matching prefix (BMP) for the destination address equal to the entry, while the latter records the BMP for the IP addresses falling between the current entry and the next one. Accordingly, the pre-computation is used to find out the BMP for each entry, that complicates table construction. Since the major bottleneck ties to the pre-computation, we will focus on its optimization.

We sort the route prefixes at first. Let S_i, E_i and l_i be the starting address, ending address and the length of the route prefix P_i, respectively, and assume $S_1 \leq S_2 \leq \cdots \leq S_N$ (N is the number of route prefixes, $1 \leq i \leq N$). If $S_i = S_j$, then $i < j$ if and only if $l_i < l_j$. With the sorted prefixes, just processing each route prefix sequentially can accomplish the calculation of the BMP.

For the route prefix P_i, both values of **equal** and **large** fields in the entry with starting address can be filled with P_i, yet they may not be the BMP. However, the longer matching prefix will be processed after P_i according to the rule of sorted prefixes. Thus both values will be overwritten with the correct prefix identifier. To process the entry with ending address, we have to consider two problems: its relative location within the processed entries and the values of both **equal** and **large** fields. We adopt two arrays, L_1 and L_2, for computation. The first one is used to store the generated entry. The other is implemented as a stack which performs **push** and **pop** operation. The entry with starting address will be inserted into L_1 directly, while the entry with ending address will be pushed into L_2 at first. To place the entry with ending address in correct position, we check whether

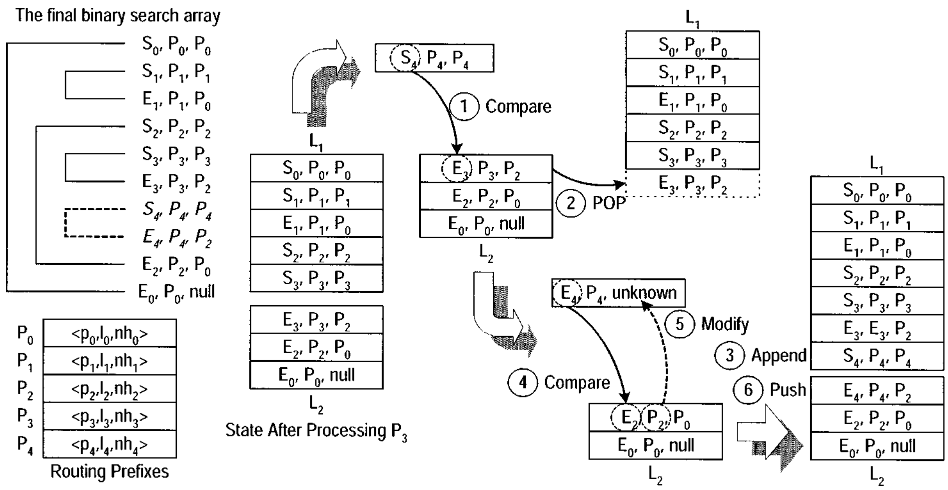

Fig. 1. NHA construction example.

the address of the top entry in L_2 is smaller than S_i. If so, we pop the top element of L_2 and append it to L_1. We repeat these steps until there is no entry in L_2 smaller than S_i. Then we must check whether the address of the rear entry in L_1 is equal to S_i or not. If yes, it means there exists a longer prefix with same starting address as described. Thus the rear entry in L_1 will be overwritten with the current one. Otherwise, the entry will be appended.

Before an entry with ending address is added to L_2, we check whether the address of the top entry in L_2 is equal to E_i or not. If yes, we update the **equal** field of the top entry with P_i for a longer matching prefix as described above. Otherwise, $\langle E_i, P_i, P_{top}\rangle$ will be pushed into L_2 where the P_{top} is the **equal** field of the top entry in L_2. This is because that the region indicated in the **large** field is occupied by a shorter prefix which forms the top entry in L_2. Consequently, the entry will be pushed into L_2. After processing all route prefixes, we pop all entries stored in L_2 and append them to L_1 in sequence. The detailed algorithm is shown as follows.

Entry Pre-Computation Algorithm
Input: N routing prefixes.
Output: The array of the processed entries.

The item SA_{top} and SA_{rear} represent the addresses of the top and the rear entries in L_1 and L_2, respectively.
Let $P = P_1, P_2, \ldots, P_{N-1}$ be the set of sorted prefixes of an input segment.
Append the entry $\langle 0, default\ route,\ default\ route\rangle$ into L_1 and push another entry $\langle 1111\cdots 1, default\ route, -\rangle$ into L_2.

For $i = 1$ to N **do**

1. For the entry $\langle S_i, P_i, P_i\rangle$ of ith routing prefix.
Check if S_i larger than SA_{top}.
2. If yes, pop out the top entry from L_2 and insert it into L_1. Repeat step 1&2 until the result of the comparison is false.
3. Check if $SA_{rear} = S_i$,
3.a If yes, overwrite the rear entry in L_1 with $\langle S_i, P_i, P_i\rangle$.
3.b Otherwise, append $\langle S_i, P_i, P_i\rangle$ into L_1.
4. Check if the $SA_{top} = E_i$,
4.a If yes, overwrite the "**equal**" field of the top entry in L_2 with P_i.
4.b Otherwise, push $\langle E_i, P_i, P_{top}\rangle$ into L_2, where P_{top} is the "**equal**" field of the top entry in L_2.

Pop out all entries from L_2 and append them into L_1.
End.

After processing all prefixes, the ordered entries will be available in the array. The time complexity of the entry pre-computation algorithm without the prefix sorting is $O(N)$. In the worst case, the number of entries is twice as the number of route prefixes. However, since the entry merging is preformed in Steps 3.a and 4.a for the entries with duplicate address, the number of entries is slightly reduced. Note that if we didn't merge the duplicate entry, it may cause ambiguous situation during the lookup process, because it is unable to tell which entry indicates the longer prefix.

In Fig. 1, we use an example to illustrate the algorithm. After processing the route prefix P_3, the top element of L_2 and rear elements of L_1 are $\langle E_3, P_3, P_2\rangle$ and $\langle S_3, P_3, P_3\rangle$, respectively. For the route prefix P_4, it firstly compares with SA_{top}. Since $SA_{\text{top}} = E_3$ is smaller than S_4, will be popped out and appended to L_1. Also, there is no smaller entry in L_2, we further check whether $SA_{\text{rear}} (= E_3)$ is equal to S_4 or not. Because they are different, the entry $\langle S_4, P_4, P_4\rangle$ will be appended to L_1. In addition, SA_{top} is not equal to E_4, thus it pushes the entry $\langle E_4, P_4, P_2\rangle$ into L_2.

TABLE I
PERFORMANCE EVALUATION WITH FIVE ROUTING TABLES

Performance Metrics	AADS	Mae-East	Paix	PacBell	Mae-West
Prefix Count	23,515 (-)	53,226 (38,816)	29,959 (3,812)	11,587 (713)	33,514 (14,072)
Memory Required (Kbytes)	566 (-)	1,217 (950)	820 (530)	487 (265)	876 (512)
Build Time (msec)	110 (-)	219 (3,900)	140 (520)	78 (230)	141 (1,440)
Worst-Case Update Time (μsec)	275	350	320	259	305
Worst-Case Lookup Time (nsec)	387 (-)	432 (330)	384 (260)	380 (200)	390 (330)
Average Lookup Time (nsec)	89	116	97	85	102

III. PERFORMANCE EVALUATION

We choose a 300-MHz Pentium II that has a 512-kbytes L2 cache and runs Windows NT for the experiment. Five routing tables obtained from the IPMA project [4] on May 15, 2000 are used. We will show the performance of the proposed scheme in the aspects of construct/update time and storage. To show the worst-case update cost, we choose the segment (first 16-bits of IP address) with the maximum number of route prefixes. Then we make 1024 copies of this segment and construct the forwarding table for them. The total storage for building the forwarding table is larger than 2 MB. This will ensure that each acquired block is fetched from the main memory, which reflects the real situation. The worst-case update time can be calculated by dividing the total elapsed time by the number of iterations (1024). We also showed the worst and the average case lookup performance with the same methodology. Since the tree construction and search algorithms are not listed in the literature, thus there might be some bias between both implementations. But we believe this would be small enough to be ignored.

The experimental results are shown in Table I. For convenience, the available numerical results from [3] are listed in the parentheses. Note that numbers measured on 200-MHz Pentium Pro in [3] have been projected to 300 MHz. First of all, the size of the forwarding table is in proportional to the size of the routing table. Obviously, the total construction time is much less than that in BSP scheme. Further, the more entries in the routing table, the greater improvement can be achieved. The worst-case lookup time in our implementation is slower than that in the previous work. This is because there are much more entries in the tested routing tables. And also, the projection of CPU speed might over-estimate the performance since the memory speed is not improved as well as the CPU speed. In [3], the authors claimed that the average and worst case update time are 20 ms and 350 ms, respectively. With our algorithm, the table update time is less than 350 μsec, which shows a significant improvement in performance.

IV. CONCLUSIONS

In this letter, we aim at enhancing the BSP for prefix longer than 16 bits with a fast forwarding-table construction algorithm. Since the update cost ties to the table construction process, we propose a fast forwarding-table construction algorithm and resolve the ambiguous lookup problem caused by duplicate entry. With the current 53 000-entry routing table of a backbone router, our algorithm can achieve more than 3600 route updates per second in the worst case. It is obvious that the proposed algorithm improves the performance significantly. Moreover, it is simple enough to fulfill the need of high-speed packet forwarding.

REFERENCES

[1] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proc. ACM SIGCOMM'97*, Cannes, France, Sept. 1997, pp. 3–14.

[2] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM'98*, San Francisco, CA, Mar. 1998.

[3] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," *IEEE/ACM Trans. Networking*, vol. 7, pp. 323–334, June 1999.

[4] Merit Networks, Inc.. Internet performance measurement and analysis (IPMA) statistics and daily reports. [Online]. Available: http://www.merit.edu/ipma/routingtable/

[5] Y. Rekhter and T. Li, "An architecture for IP address allocation with CIDR,", RFC 1518, Sept. 1993.

[6] M. Waldvogel, G. Vargnese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in *Proc. ACM SIGCOMM'97*, Cannes, France, Sept. 1997, pp. 25–36.

[7] S. Bradner, "Next generation routers. Overview," *Proc. Networld Interop.*, 1997.