Routing Table Compaction in Ternary CAM

THESE TECHNIQUES FOR REDUCING THE SIZE OF ROUTING TABLES STORED IN TERNARY CONTENT-ADDRESSABLE MEMORY RESULT IN DECREASED COST, POWER CONSUMPTION, AND THERMAL DISSIPATION.

Huan Liu
Stanford University

To determine how to forward a packet, an Internet router must perform routing lookup on the destination IP address. Since the adoption of classless interdomain routing in 1993, routing an incoming packet requires that the router find the longest routing prefix that matches the destination IP address.

Researchers have proposed various software-based schemes to accelerate the lookup function.[1-3] However, all of these approaches require at least four to six memory accesses. With the requirement for higher throughput, the latency and bandwidth of modern memory architecture severely limit the number of memory accesses a system designer can afford. Clearly, the software-based solutions do not easily scale up to 10-Gbps processing and beyond.

Of the several hardware-based solutions proposed, some use dedicated special hardware[4,5] and others use commercially available content-addressable memory (CAM).[6,7]

CAM allows simultaneous comparison between all indexes and the key (the destination IP address), and the entry corresponding to the matched index can be obtained directly. CAM's main advantage is that search time is bounded by a single memory access; thus, it can guarantee high lookup throughput. There are two types of CAM: binary, where each bit position stores only 0 or 1, and ternary, where each bit position can store 0, 1, or don't care. Binary CAM allows only fixed-length comparisons, so it isn't directly suitable for longest-prefix matching. A possible solution is to store prefixes of varying length in a separate binary CAM, then design external logic to pick the longest matched entry from all matched CAM chips.[8]

Ternary CAM (TCAM) could solve the longest-prefix-matching problem more directly.[7] In addition to the index, TCAM also stores a separate mask for each entry. The mask specifies which bits in the index are active, thereby specifying the variable-length prefix. Table 1 is an example routing prefix table stored in TCAM.

Commercially available CAM costs much more than conventional memory, and TCAM is even more expensive. In addition, they consume more power and dissipate more heat, posing a system design challenge. Therefore, it would be advantageous to compact the routing table so that the system could use fewer CAM chips. Even for a single CAM chip, a routing table compaction scheme can help reduce power consumption and heat dissipation. Because the number of routing prefixes is increasing steadily, a routing table compaction scheme can also help contain the routing table size explosion. I propose two techniques to compact routing tables stored in TCAM.

0272-1732/02/$17.00 © 2002 IEEE

Table 1. A prefix table stored in TCAM.

No.	Prefix	Mask	Next hop port
P_1	10011100	11111100	7
P_2	10001100	11111100	7
P_3	11011100	11111100	7
P_4	10001000	11111000	5
P_5	11010000	11110000	4
P_6	11110000	11110000	7
P_7	10010000	11110000	4

Table 2. Routing table characteristics.

IXP	No. of routes	No. of prefixes
Mae East	36	23,554
Mae West	40	32,139
Pacific Bell	19	38,791
Aads	37	15,906
Paix	26	29,195

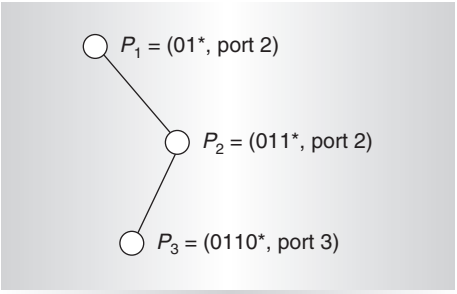

Figure 1. In this pruning example, the routing table is organized as a tree structure. Prefix P_1 is an identical parent of P_2, and P_2 is a parent of P_3. P_2 is a redundant prefix that can be removed without affecting routing functionality.

Prefix compaction

The number of possible routes (next hop) in a routing table is typically small because only a limited number of interface cards fit into the router chassis. In contrast, there are typically many routing prefixes—in the range of several thousand. Table 2 shows the number of routes and routing prefixes in several backbone routers (http://www.merit.edu/ipma) at major US Internet Exchange Points (IXP). The routing table has at least 500 times more prefixes than routes. It is possible to exploit this disparity to compact a routing table.

Pruning

The pruning technique eliminates redundant routing prefixes. First I will define some terms. I use $|P_a|$ to denote the length of prefix P_a, and I use $P_{a,i}$ to denote the ith bit of the prefix, where $P_{a,1}$ is the most significant bit and $P_{a,|Pa|}$ is the least significant bit. A prefix P_a is the parent of prefix P_b if the following three conditions hold:

1. $|P_a| < |P_b|$.
2. $P_{a,i} = P_{b,i}$ for all $1 \le i \le |P_a|$.
3. There is no prefix P_c such that $|P_a| < |P_c| < |P_b|$, and $P_{c,i} = P_{b,i}$ for all $1 \le i \le |P_c|$.

Intuitively, the parent of prefix P_b is the longest prefix that matches the first few bits of P_b. A parent P_a of prefix P_b is an identical parent if P_a translates to the same route as P_b—that is, packets matching both prefixes will be routed to the same next hop.

The idea of pruning is fairly simple. If P_a is an identical parent of P_b, then P_b is a redundant routing prefix. To understand this, assume the longest prefix matched for an IP address is P_b; by definition, the IP address will match P_a as well. With P_b removed from the routing table, P_a becomes the longest matched prefix. Because they both translate to the same route, removing P_b makes no difference. Note that this technique is general enough that it applies to any routing lookup algorithm, regardless of how the routing table is stored. Figure 1 shows an example of pruning.

Mask extension

The second technique exploits TCAM hardware's flexibility. The mask for a routing prefix stored in TCAM consists of ones (the same number of ones as the prefix length) followed by all zeros. However, TCAM allows the use of an arbitrary mask, so that the bits of ones or zeros needn't be continuous. I call this technique mask extension because it extends the mask to be any arbitrary combination of ones and zeros.

A simple example helps to describe the mask extension technique. In Table 1, P_1 and P_2 both correspond to the same route, port 7. It's possible to combine the two prefixes into a single entry, with the prefix set to 10001100

Table 3. Compacted table using mask extension.

No.	Prefix	Mask	Next hop port
P_1 and P_2	10001100	11101100	7
P_1 and P_3	10011100	10111100	7
P_4	10001000	11111000	5
P_5 and P_7	10010000	10110000	4
P_6	11110000	11110000	7

```
Compact_routing_table ( )
    For all prefix length l
        For all possible next hop n
            C(l, n) = minimize( P(l, n), nil );
            insertIntoCAM( C(l, n) );
```

Figure 2. Pseudocode to compact a routing table using mask extension.

and the mask set to 11101100, as shown in Table 3. The zero at bit 4 (counting from the left) in the mask prevents comparison at that bit and allows matching of P_1 and P_2 in the same entry. Using only the mask extension technique, Table 3—the compacted version of the original routing table (Table 1)—has been reduced from seven entries to five.

The mask extension technique reduces to a logic minimization problem. In the discussion, I use *cube* to refer to the combined single entry for several prefixes, and *cover* to refer to the set of cubes that cover all prefixes. The problem then becomes: Given a set of prefixes with the same length and same route, find a minimal cover.

Logic minimization is an NP-complete problem, so there is little hope of finding an efficient, exact algorithm. Fortunately, there is a fast, proven, and very efficient heuristic algorithm—Espresso-II[9]—that produces a near-optimal solution with finite computing resources.

To show how the mask extension technique could be implemented, I will first define some terms.

- $P(l, n)$: The set of prefixes that have length l and next hop port n.
- $C(l, n)$: The set of cubes that cover $P(l, n)$. This is the result of logic minimization.
- S_{on}: The "on set" for logic minimization.
- S_{dc}: The "don't care set" for logic minimization.
- $F(P)$: The set of cubes that cover prefix P.

Figure 2 shows the pseudocode to compact a routing table using mask extension. The routine finds the set of routing prefixes with the same prefix length and route, then uses the Espresso-II logic minimization algorithm to compute the minimal cover. The **`minimize()`** routine is the Espresso-II logic minimization algorithm described by Brayton et al.[9] The routine takes two arguments: the on set to be minimized, which is the set of prefixes that must be used to compute the cover, and the don't care set, which is the set of prefixes that could be used to compute a better cover. The **`insertToCAM()`** routine stores the compacted table into a TCAM array for routing lookup.

Incremental update

The routing table is hardly static. In fact, there could be hundreds of updates (inserts/withdraws) per second.[10] Most routing updates are route flaps—that is, the same prefix is added and then removed repeatedly in quick succession. It is straightforward to reduce the number of actual updates by keeping a buffer of recent route update announcements and updating only the end result. This eliminates most route flaps. Still, tens of updates per second may be required in backbone routers, so a fast update algorithm is necessary. Incremental update for the pruning technique is quite straightforward. However, incremental update for the mask extension technique is nontrivial and requires a fast update algorithm.

Insertion

When a new prefix arrives, I could reminimize the whole set $P(l, n)$ along with the new prefix, although the computation requirement would be extensive. Another approach would be to insert the new prefix directly into the TCAM array, because a prefix is itself a cube, although not necessarily the largest cube. This simple algorithm will eventually reduce the compaction ratio; it will produce almost no area savings at all.

Instead of these two naïve approaches, I will describe a heuristic minimization algorithm to facilitate fast incremental insertion. When a new prefix P is inserted, I compute a mini-

```
Insert_prefix (P)
    S_on = { P };
    S_dc = C(l, n);
    C_p = minimize(S_on , S_dc);
    For each C in C(l, n)
        /* see if C is a subset of C_p */
        If ( subset(C, C_p )
            /* remove C if duplicate */
            remove(C, C(l, n) );
            /* also remove it from TCAM array */
            removeFromCAM( C);
    C(l, n) = C(l, n) + {C_p };
    insertIntoCAM( {C_p } );
```

Figure 3. Rather than the new prefix, the incremental insertion algorithm inserts a minimal cover into the TCAM array.

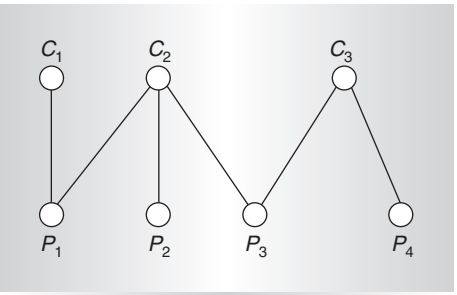

Figure 4. Example cube and prefix relationship. A direct line between a cube and a prefix means the cube covers the corresponding prefix.

mal cover C_p, using P as the on set of minimization and the existing compacted table as the don't care set. Instead of new prefix P, I insert minimal cover C_p into the TCAM array. Because the C_p insertion could make existing entries in the TCAM array redundant, I examine each existing entry in turn and remove entries completely covered by C_p. The pseudocode for the incremental insertion algorithm appears in Figure 3.

For example, if I insert a new entry P = (110011--, 7) into the routing table (Table 1), S_{on} = {110011--}, S_{dc} = {100-11--, 1-0111--}, and, after minimization, C_p = 1-0-11--. The subsequent redundancy check will remove 100-11-- and 1-0111-- from TCAM because they are completely covered by the new cube C_p. The resulting $C(6, 7)$ is {1-0-11--}.

Withdrawal

The algorithm for removing a prefix from the routing table is more complex because several cubes could cover the prefix. I must remove all cubes covering the prefix and recalculate a minimum cover from the affected prefixes. Figure 4 provides an example. C_1, C_2, and C_3 are cubes, and P_1, P_2, P_3, and P_4 are prefixes.

If P_3 needs to be removed, then C_2 and C_3 must be removed. As a result, P_2 and P_4 no longer have any cover, so they must be included in the computation for new cover. Note that P_1 isn't affected, because although C_2 is removed, C_1 still covers P_1. The incremental removal algorithm searches for prefixes no longer covered by cubes and includes them in the computation for new cover. The pseudocode for the incremental removal algorithm appears in Figure 5. As in the incremental insertion algorithm, this algorithm requires eliminating redundant cubes.

For example, if we are removing P_1 from Table 1, S_{on} = {100011--, 110111--}, S_{dc} = *nil*, and, after minimization, *set* = {100011--, 110111--}. The subsequent loop cannot find a redundant cube, so the resulting $C(6, 7)$ is {100011--, 110111--}.

```
Withdraw_prefix (P)
    S_on = nil ;     /* initialize S_on */
    /* search for prefix affected by removal of P */
    For each C in F(P)
        remove(C, C(l, n) );      /* remove C from the set */
        removeFromCAM(C);  /* also remove from TCAM */
        For each P' covered by C
        remove(C, F(P') );  /* remove C as cover for P' */
        If ( F(P') = nil )
            S_on = S_on + {P'};
    S_dc = C(l, n);
    set = minimize(S_on, S_dc )

    /* remove redundant cubes */
    For each C in C(l, n)
        /* if C is a subset of set, remove it */
        If ( subset(C, set) )
            remove(C, C(l, n) );
            removeFromCAM(C);
    C(l, n) = C(l, n) + set;
    insertIntoCAM( set );
```

Figure 5. The incremental withdraw algorithm must find prefixes no longer covered by cubes and include them in the computation for new minimal cover.

Table 4. Routing table compaction result.

IXP	Original size (prefixes)	Size after pruning	Savings after pruning (%)	Size after pruning and mask extension	Savings after pruning and mask extension (%)	Runtime(s)
Mae East	23,554	17,791	24.5	13,492	42.7	14.20
Mae West	32,139	24,741	23.0	18,105	43.7	49.30
Pacific Bell	38,791	28,481	26.6	20,166	48.0	77.00
Aads	29,195	21,857	25.1	16,057	45.0	44.10
Paix	15,906	11,828	25.6	8,930	43.9	8.26

Experimental result

I implemented these algorithms and evaluated their performance on several routing tables in backbone routers located at major IXPs. I used the routing tables captured by the Internet Performance Measurement and Analysis project (http://www.merit.edu/ipma) on 7 March 2001.

I first used pruning to remove redundant routing prefixes, then the mask extension technique to further reduce table size. The Espresso-II algorithm used in the mask extension step has several options to control minimization quality and runtime. The "exact" option enables finding the minimal number of cubes, though not necessarily the minimal number of literals. (Any bit having the value 1 or 0, rather than don't care, counts as one literal.) This is the optimum solution for the problem because each cube corresponds to one TCAM entry; minimizing the number of cubes is equivalent to minimizing the whole routing table's size. Reducing the number of literals doesn't further compact the routing table. Theoretically, the runtime could be quite high for exact minimization. In fact, if the pruning technique isn't used, the runtime for the mask extension step takes twice as long when exact minimization is enabled. Pruning first made the routing table substantially smaller, and different minimization options didn't noticeably affect runtime. Therefore, I show only results with exact minimization enabled, because it produces a slightly better compaction result at little extra cost.

Table 4 shows the original table size in number of prefixes, its size after pruning, and its size after pruning and mask extension, along with the percentage savings and the computation time. Pruning alone consistently reduces the routing table's size by roughly 25 percent. With mask extension also applied, the overall savings ranged from 42.7 to 48 percent. Mask extension saves roughly an additional 20 percent. Although not shown, mask extension alone (without pruning) can reduce table size between 27.3 and 30.4 percent, but runtime is greater because of the larger input for logic minimization. The Espresso-II algorithm exhibits exponential runtime with respect to the input size, so pruning first reduces the runtime of the mask extension step. I measured the overall runtime shown in Table 4 on a Pentium III 500-MHz PC platform. The runtime for large routing tables becomes much greater because of Espresso-II's exponential behavior.

One way to further increase the savings is through prefix expansion.[11] This reduces the number of possible prefix lengths to a fixed small number, and increases the size of the on set at each chosen prefix length. As a result, there's an increased chance of better logic minimization and therefore a smaller compacted routing table. The runtime required is substantially greater, however, because of the huge input, so the practical use of prefix expansion is limited.

Large routing tables could require a lot of time for compaction. Many routing prefixes have the same length. For example, for the Pacific Bell routing table, most prefixes are 24 bits long. In particular, there are 11,634 24-bit-long prefixes for a particular next hop port. Since the Espresso-II algorithm examines all 11,634 prefixes to produce the cover, a lot of computation time is spent on 24-bit-long prefixes. It's possible to reduce the runtime by breaking the input into smaller sections, minimizing them individually, and then combining the results for final minimization.

Because of the mask extension technique's

long runtime, it should run only once, at system initialization. Thereafter, the incremental update algorithm should handle routing updates. The incremental update for the pruning technique is trivial, so I don't evaluate its performance here. Instead, I use only mask extension to compact the routing table and then apply the incremental update algorithm to evaluate its performance. I used a 12-hour routing update trace captured from the Mae East IXP. The trace contains nearly 35,000 updates. I plotted the original and the compacted routing table sizes after each insertion and withdrawal. The result appears in Figure 6. Clearly, after numerous updates the compacted table's size still closely follows that of the original table, with the gap remaining fairly constant. Thus, the incremental update algorithm can maintain the savings. In fact, at the end of the trace, the gap between the two lines increases by nearly 300 entries, because the larger table at the end presents better optimization opportunities. In my implementation, the average runtime for each update is 22 ms on a Pentium III 500-MHz processor—fast enough to support up to 50 updates per second. Combined with the route flaps buffering mechanism, the incremental update algorithm can handle typical updates in the Internet backbone.

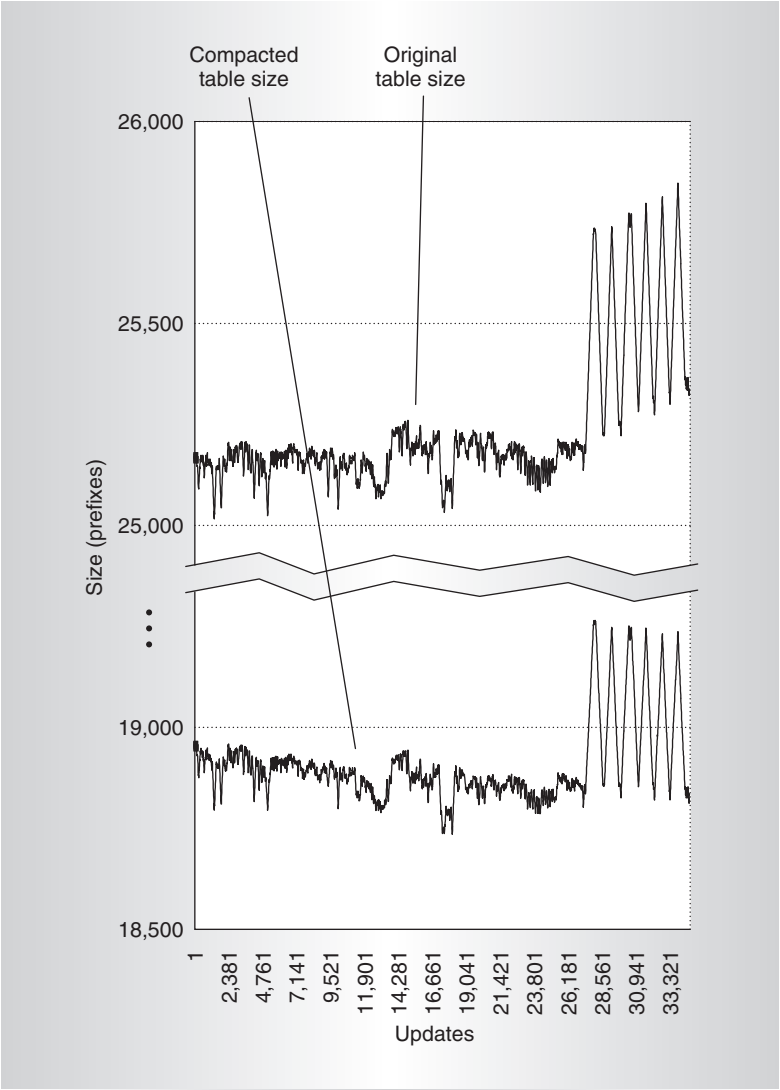

Figure 6. Original and compacted table size after incremental updates. (Note that a portion of the scale has been omitted.)

Because of the software-based lookup algorithm's speed limitation, TCAM is helping to solve the longest-prefix-matching lookup problem in state-of-the-art, high-speed router design. The space savings in the simulation result translates directly into cost savings, because designers can use either fewer TCAM chips or a smaller capacity TCAM. In addition, this approach results in less power consumption and less heat dissipation. The techniques can also help routers effectively cope with the routing prefix explosion. MICRO

References

1. S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-tries," *IEEE J. Selected Areas Comm.*, vol. 17, no. 6, June 1999, pp. 1083-1092.
2. M. Waldvogel et al., "Scalable High-Speed IP Routing Lookups," *Computer Comm. Rev.*, vol. 27, no. 4, Oct. 1997, pp. 25-36.
3. M. Degermark et al., "Small Forwarding Tables for Fast Routing Lookups," *Computer Comm. Rev.*, vol. 27, no. 4, Oct. 1997, pp. 3-14.
4. P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. IEEE Infocom*, vol. 3, IEEE CS Press, Los Alamitos, Calif., 1998, pp. 1240-1247.
5. M. Kobayashi, T. Murase, and A. Kuriyama, "A Longest Prefix Match Search Engine for Multi-Gigabit IP Processing," *Proc. IEEE Int'l*

Conf. Comm., June 2000, IEEE Press, Piscataway, N.J.
6. T.B. Pei and C. Zukowski, "VLSI Implementation of Routing Tables: Tries and CAMs," *Proc. IEEE Infocom*, vol. 2, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 515-524.
7. A. McAuley and P. Francis, "Fast Routing Table Lookup Using CAMs," *Proc. IEEE Infocom*, vol. 3, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 1382-1391.
8. T. Hayashi and T. Miyazaki, "High-Speed Table Lookup Engine for IPv6 Longest Prefix Match," *Proc. IEEE Globecom*, vol. 2, IEEE Press, Piscataway, N.J., 1999, pp. 1576-1581.
9. R.K. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Boston, 1984.
10. C. Labovitz, G.R. Malan, and F. Jahanian, "Internet Routing Instability," *IEEE/ACM Trans. Networking*, vol. 6, no. 5, Oct. 1998, pp. 515-528.
11. V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Computer Systems*, vol. 17, no. 1, Oct. 1999, pp. 1-40.

Huan Liu is a PhD candidate in electrical engineering at Stanford University. His research interests include high-speed switch and network processor design. Liu received BS degrees in both physics and computer science from the Harbin Institute of Technology, China, and an MS in computer science from the University of California, Los Angeles.

Direct questions or comments about this article to Huan Liu, Department of Electrical Engineering, Stanford University, Stanford, CA 94305; huanliu@stanford.edu.