FAST UPDATING ALGORITHMS FOR TCAMS

ONE POPULAR HARDWARE DEVICE FOR PERFORMING FAST ROUTING LOOKUPS AND PACKET CLASSIFICATION IS A TERNARY CONTENT-ADDRESSABLE MEMORY (TCAM). WE PROPOSE TWO ALGORITHMS TO MANAGE THE TCAM SUCH THAT INCREMENTAL UPDATE TIMES REMAIN SMALL IN THE WORST CASE.

Devavrat Shah
Pankaj Gupta
Stanford University

Internet routers look up the destination address of an incoming packet in its forwarding table to determine the packet's next hop on its way to the final destination. This routing lookup operation takes place on each arriving packet by every router in the path that the packet takes from its source to the destination.

The adoption of classless interdomain routing (CIDR)[1] in 1993 required a routing lookup to perform a longest prefix match operation. A router maintains a set of destination address prefixes in a forwarding table. Given a packet, the operation finds the longest prefix in the forwarding table that matches the first few bits of the packet's destination address.

To provide enhanced services such as packet filtering, traffic shaping, policy-based routing, and so on, routers also must be able to recognize flows. A flow is a set of packets that obey some rule, also called a policy, on the packet's header fields. These fields include source and destination Internet protocol addresses, source and destination port numbers, protocol, and others. For instance, all packets with a specified destination IP address and specified source IP address may be defined by a rule to form a single flow.

A collection of rules is called a policy database or a classifier. Identification of the flow of an incoming packet is called packet classification and is a generalization of the routing lookup operation. Packet classification requires the router to find the best-matching rule among the set of rules in a given classifier that match an incoming packet. A rule may specify a prefix, range, or a simple regular expression for each of several packet header fields. An arriving packet's header may satisfy the conditions of more than one rule—in which case the rule with the highest priority determines the flow of the arriving packet.

At the time of this writing, improvements in optical communication technologies such as dense wavelength-division multiplexing (DWDM) have resulted in continually increasing link speeds up to 40 Gbps per installed fiber. However, routers have been largely unable to keep up at the same pace; a maximum of 10 Gbps (OC192) ports are available now. One main reason for this is the relatively complex packet processing required at each router.

As a result, the problems of routing lookup and packet classification have recently received considerable attention, both in academia and the industry. See, for example, the literature for solutions to the routing lookup problem[2-7] and for solutions to the packet classification problem.[8-13] Many of these reports have indicated the difficulty of the general multidimensional packet classification problem in the worst case.

Hardware realizations of algorithmically simpler solutions such as linear or fully associative searches have found favor in some commercial deployments. A popular device is a

0272-1732/01/$10.00 © 2001 IEEE

special type of fully associative memory: a ternary content-addressable memory (TCAM). Each cell in a TCAM can take three logic states: 0, 1, or don't-care X. A CAM allows a fully parallel search of the forwarding table or a classifier database. The ternary capability lets the TCAM store wild cards and variable-length prefixes by storing don't-cares. Lookups are performed in a TCAM by storing forwarding table entries in order of decreasing prefix lengths and choosing the first entry among all the entries that match the incoming packet's destination address. Packet classification is carried out similarly by storing classifier rules in order of decreasing priority.

The need to maintain a sorted list makes incremental updates slow in a TCAM. If N is the total number of prefixes to be stored in a TCAM having M entries, naive addition of a new entry can result in the need to move $O(N)$ TCAM entries to create the space required to add the entry at a particular place in the TCAM to maintain the sorted order. Alternatively, some TCAM entries can be intentionally left unused in anticipation of future additions. However, this leads to wasted space and underutilization of the TCAM. Besides, the worst case still remains $O(N)$.

We were motivated by the desire to simultaneously achieve fast incremental updates as well as full use of the TCAM. With this objective, we describe worst-case algorithms (one specific for route lookups, and the other suitable for both lookups and classification) that achieve the optimal number of a TCAM's operations (such as move/write/read) required for an incremental update. The algorithms are online in the sense that they perform operations on memory as update requests arrive, instead of batching several update requests.

In particular, we show that, if L is the width of the destination address field (L equals 32 in IPv4, and 128 in IPv6), no more than $L/2$ memory operations are required. This algorithm is proved to be optimal; that is, it performs no worse than any other algorithm in the worst case that keeps the list of forwarding table entries in order of decreasing prefix lengths. This compares favorably with the L memory operations in the memory management schemes recommended by some TCAM vendors (see Sibercore Technologies, www.sibercore.com/scan01_cidr_p03.pdf current Feb. 2000). See also our later discussion.

It turns out that it isn't necessary to keep all the forwarding table entries in the order of decreasing prefix lengths; instead, only overlapping prefixes need to be in this order. Two prefixes overlap if one is a prefix of the other; for example, 01* overlaps with 0101*, but not with 001*. This observation is used in the second algorithm. Although not proved, this algorithm seems to be optimal in the number of worst-case memory operations required to handle a forwarding table update. We also describe how this algorithm and the results for routing lookups extend to packet classification.

To the best of our knowledge, there's no previous work that attempts to (algorithmically) optimize updates on a TCAM. Most TCAM vendors live with an $O(N)$ worst-case update time solution. Some attempt to provide a hardware maximum function that computes the maximum of the prefix lengths (or priorities) of all matching entries, hence eliminating the requirement of keeping the table entries sorted. However, computing the maximum of $O(M)/\log_2 M$-bit numbers is expensive in current technology in terms of logic area and speed. (M is around 16K to 64K presently). This is likely to worsen in the future as TCAMs scale to greater densities. One recent paper[14] uses circuit-level optimizations for fast updates at the cost of slower search time and lower memory density.

Note that while the algorithms we discuss have been mentioned in the context of a parallel-search TCAM, they are equally applicable to other algorithms that keep a sorted list of forwarding table entries or classifier rules, such as hardware realizations of a linear search algorithm.

Longest prefix matching using TCAMs

IP addresses are written in the dashed quad notation, for instance, 103.23.3.1, representing the four bytes of an IPv4 destination address separated by dots. An entry in a router's forwarding table is a pair: <*route-prefix,nextHop*>. A *route-prefix*, or simply a prefix, is represented like an IP address but may have some trailing bits treated as wild cards; this denotes the aggregation of several 32-bit destination IP addresses. For example, the aggregation of 256 addresses 103.23.3.0

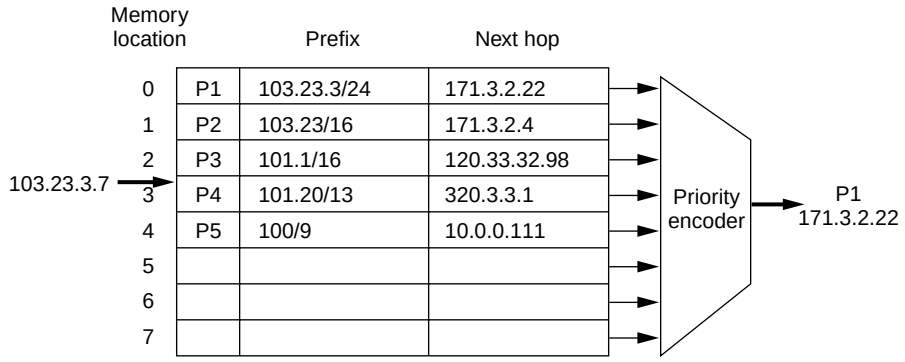

Figure 1. Longest prefix matching using a TCAM.

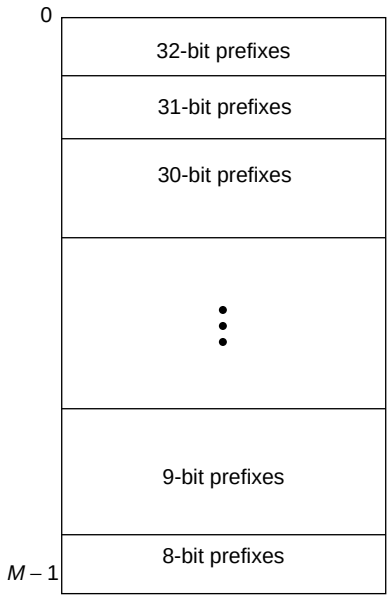

Figure 2. General configuration of a TCAM used for longest prefix matching. No prefixes of a length less than 8 bits are shown because they are typically not found in forwarding tables.

through 103.23.3.255 is represented by the prefix 103.23.3/24, where 24 is the length of the prefix, and the last 8 bits are wild cards. Other examples of prefixes are 101/8, 54.128/10, 38.23.32/21, and 200.3.41.1/32. *nextHop* is the IP address of a router or end host that is a neighbor of this router.

Given an incoming packet's destination address, a routing lookup operation finds the entry with the longest—that is, the most specific—of all the prefixes matching the first few bits of the incoming packet's destination address. It then forwards the incoming packet to this entry's next-hop address.

This longest prefix matching operation is performed in a TCAM by storing entries in decreasing order of prefix lengths. The TCAM searches the destination address of an incoming packet with all the prefixes in parallel. Several prefixes (up to $L = 32$ in the case of IPv4 lookups) may match the destination address. A priority encoder logic then selects the first matching entry—the entry with the matching prefix at the lowest physical memory address. Figure 1 shows an example.

Figure 2 shows the general configuration for storing N prefixes in a TCAM with M memory locations. We refer to the set of all prefixes of length j as P_j. We also assume a memory manager software that arranges prefixes in the desired order and sends appropriate instructions to the TCAM hardware.

Forwarding tables in routers are dynamic; prefixes can be added or deleted as links go up or down due to changes in network topology. These changes can occur at the rate of approximately 100 to 1,000 prefixes per second.[15] While this is slow in comparison to the packet lookup rate (which is on the order of millions of packets per second), it's desirable to obtain quick TCAM updates. Slow updates may cause incoming packets to be buffered while an update operation is being carried out. This is undesirable for many applications because it may cause head-of-line blocking and require a large buffer space separate from the main packet buffer memory in the router. Indeed, some TCAM vendors (see Netlogic Microsystems at www.netlogicmicro.com) use a single-cycle update time for a big competitive advantage. Hence, it's desirable to keep the incremental update time as small as possible.

Forwarding table updates complicate keeping the list of prefixes in the TCAM in sorted order. This issue is best explained with the example in Figure 1. Assume that a new prefix 103.23.128/18 is to be added to the forwarding table. It must be stored between prefixes 103.23.3/24 (P1) and 103.23/16 (P2), currently at memory locations 0 and 1 to maintain

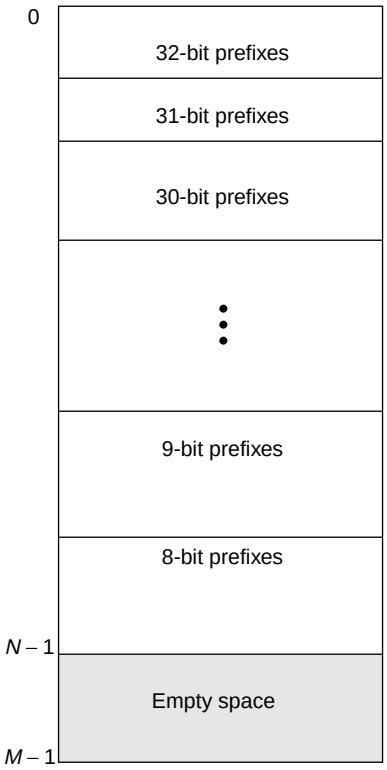

Figure 3. This naive solution keeps the free space pool at the bottom of memory.

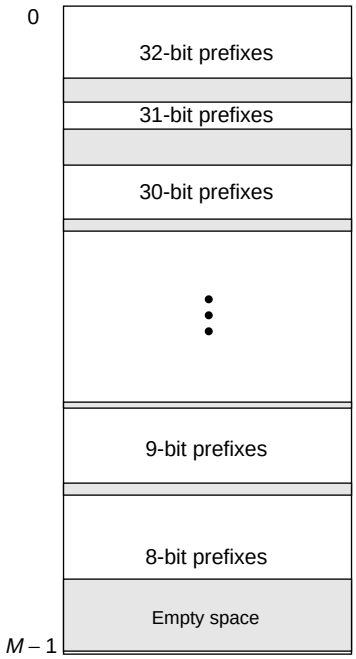

Figure 4. This solution improves the average case update time by keeping empty spaces interspersed with prefixes in the TCAM.

the sorted order. However, there's a problem since there's no empty space at that location. There can be several ways to handle this issue.

The TCAM manager can keep the free space pool (containing all unused TCAM entries) at one end of the TCAM, say at the bottom, as shown in Figure 3. A naive solution would shift prefixes P2 to P5 downward in memory by one location each, thus creating an empty space between P1 and P2 to store the new prefix. This has worst-case time complexity $O(N)$, where N is the number of prefixes in the TCAM of size M, and is clearly expensive. For instance, if N = 64,000, it will take 1.2 milliseconds (assuming one memory write operation can be performed in a 20-ns clock cycle) to complete one update operation. This is too slow for a lookup engine that completes one lookup in 20 ns because a large packet buffer would be required to store incoming packets while an update is being completed.

In anticipation of additions and deletions of prefixes, the TCAM may keep a few empty memory locations at all X nonempty memory locations, as shown in Figure 4. The average case update time improves to $O(X)$ but degenerates to $O(N)$ if the intermediate empty spaces are filled up. This solution also wastes precious CAM space.

The following solution is based on the observation that two prefixes of the same length don't need to be in any specific order. This means that if j is larger than k, all prefixes in the set P_j must appear before those in the set P_k, but prefixes within set P_j may appear in any order. Hence, there's only a partial ordering constraint between all prefixes (as opposed to a complete ordering constraint in the naive solution). We call this constraint the prefix-length ordering constraint.

This observation leads to an algorithm, referred to here as the L-algorithm, that can create an empty space in a TCAM in no more than L memory shifts (recall that L = 32), as shown in Figure 5 (next page). The average case can be improved again by keeping some empty spaces that are not all at the bottom of the TCAM. A later section proposes an optimal algorithm, PLO_OPT, that brings down the worst-case number of memory operations

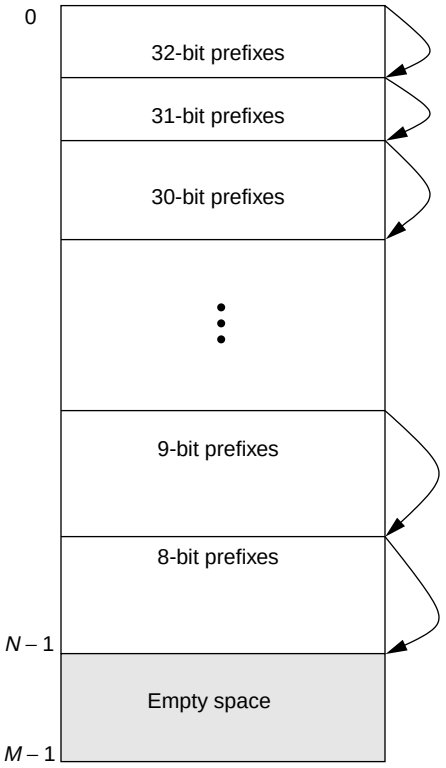

Figure 5. The prefix-length ordering constraint enables an empty memory location to be found in most $L = 32$ memory shifts.

per update to $L/2$.

The prefix-length ordering constraint is also more restrictive than that required for a correct longest prefix matching operation using a TCAM. In Figure 1, while prefix 103.23.3/24 (P1) needs to be at a lower memory address than prefix 103.23/16 (P2) at all times, it can be stored anywhere in the TCAM with respect to prefixes P3, P4, and P5. This is because P1 doesn't overlap with prefix P3 or P4 or P5. That is, no incoming destination address can match both P1 and P3, or P1 and P4, or P1 and P5. Hence, the constraint on the ordering of the prefixes in a TCAM can now be relaxed to only overlapping prefixes. Since two prefixes overlap if one is fully contained inside the other, there's an ordering constraint between two prefixes p_i and p_j if and only if one is a prefix of the other.

If all prefixes were to be visualized as being stored in a trie data structure, only prefixes that lie on the same chain (the path from the root to a leaf node) of the trie need to be ordered.

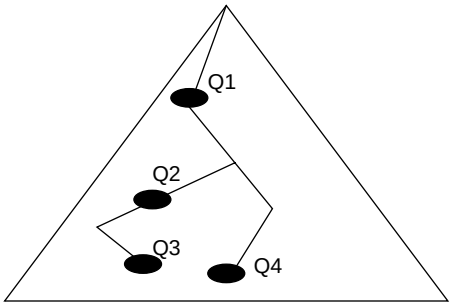

Figure 6. Illustration of the chain-ancestor ordering constraint. There are two maximal chains in this trie: one comprises Q1, Q2, and Q3; the other comprises Q1 and Q4.

For example, as shown in Figure 6, prefixes Q3, Q2, and Q1 must appear in order since they lie on the same chain. Prefix Q4 can be stored anywhere with respect to Q2 and Q3, but it must be stored at a lower memory location than Q1. We refer to this constraint as the chain-ancestor ordering constraint. A later section proposes an algorithm, CAO_OPT, that exploits this relaxed constraint to decrease the worst-case number of memory operations per update to $D/2$. Here, D is the maximum length of any chain in the trie. D is usually small (at most 5) for even large backbone forwarding tables, hence, this algorithm achieves worst-case updates in a few clock cycles.

Prefix-length ordering constraint algorithm

The basic idea of the PLO_OPT algorithm is to keep all the unused entries in the center of the TCAM. The arrangement (shown in Figure 7) is such that the set of prefixes of length L, $L - 1$, ..., $L/2$ are always above the free space pool, and the set of $L/2 - 1$, $L/2 - 2$, ..., 1 prefixes are always below the free space pool. Addition of a new prefix would have to swap at most $L/2$ memory entries to obtain an unused memory entry. Deletion of a prefix is exactly the reverse of addition, moving the newly created space back to the center of the TCAM. To support the update operations, the algorithm uses a trie data structure to keep track of the prefixes stored in the TCAM.

The average case update time can again be improved to better than $L/2$ by keeping some unused entries near each set P_i, as was done in Figure 4. The worst-case number of memory operations is now at least $L/2$ and can become

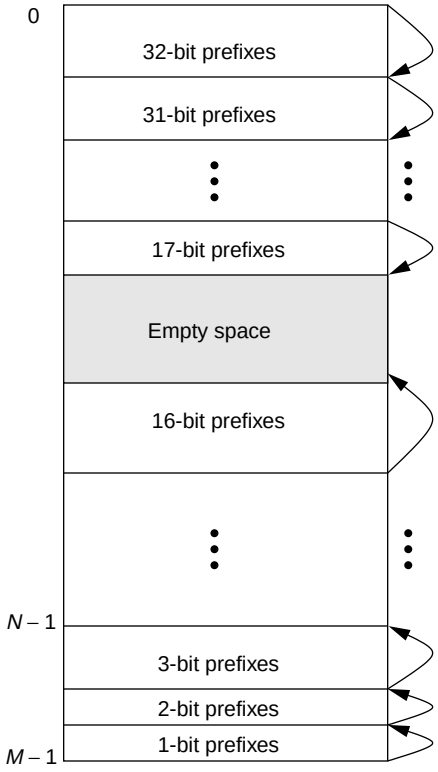

Figure 7. The PLO_OPT algorithm keeps all the unused TCAM entries in the center of the TCAM such that all prefixes longer than 16 bits are above the empty space, and all prefixes shorter than 16 bits are below the empty space at all times.

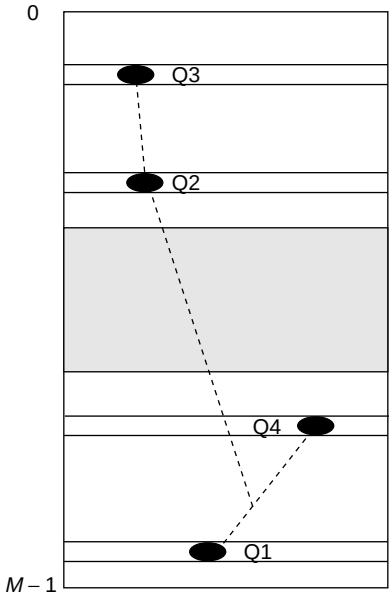

Figure 8. Memory assignment of prefixes in Figure 6 under the chain-ancestor ordering constraint. Also shown is the logical inverted trie.

even higher. The distribution of the number of unused entries to be kept around P_i depends on the distribution of updates and is therefore difficult to determine a priori. Possible heuristics for placement of an empty space include a uniform distribution and a distribution learned from recently observed update requests.

The PLO_OPT algorithm can be proved to be an optimal online algorithm under the prefix-length ordering constraint. In other words, no algorithm that is unaware of future update requests can perform better than the PLO_OPT algorithm under the prefix-length ordering constraint.

Algorithm for chain-ancestor ordering constraint

Before we describe the CAO_OPT algorithm, we need to clarify some terminology.

- *LC*(*p*) is the longest chain comprising prefix *p*.
- *len*(*LC*(*p*)) is length of (number of prefixes in *LC*(*p*)).
- *rootpath*(*p*) is the path from the trie root node to node *p*.
- *ancestor of p* is any node in *rootpath*(*p*).
- *prefix-child of p* is a child node of *p* that has a prefix.
- *hcld*(*p*) is highest prefix-child of *p*; that is, among the children of *p,* the node that has the highest memory location in the TCAM.
- *HCN*(*p*) is the chain comprising ancestors of *p*, prefix *p* itself, *hcld*(*p*), *hcld*(*hcld*(*p*)), and so on; that is, a descendant node of *p* is in *HCN*(*p*) if it's the highest prefix-child of its ancestor.

The CAO_OPT algorithm also keeps the free space pool in the center of the TCAM while maintaining the chain-ancestor ordering among the entries in the TCAM. Hence, a logical inverted trie can be superimposed on the prefixes stored in the TCAM. For example, the prefixes in Figure 6 may be stored as shown in Figure 8, with the logical inverted trie shown in dashed lines. The basic idea is to arrange the

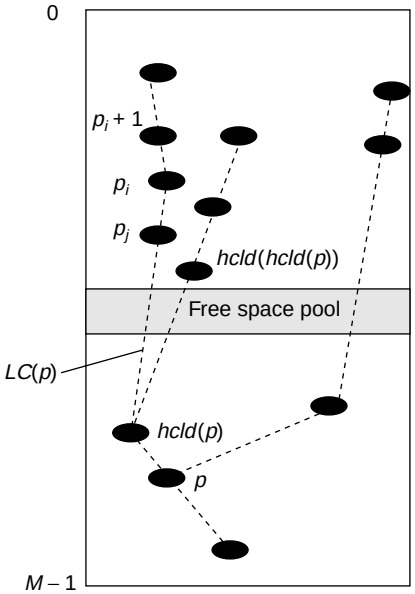

Figure 9. The distribution of chains in the TCAM under the chain-ancestor ordering constraint. Every prefix p is at a distance less than or equal to $\lceil D/2 \rceil$ prefixes for the free space pool, where D equals $len(LC(p))$.

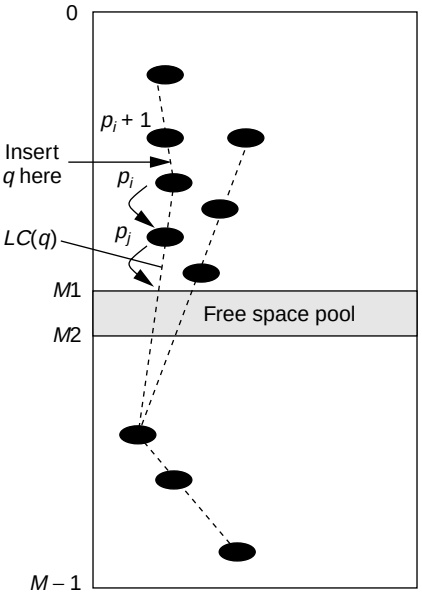

Figure 10. The way an insertion proceeds in the CAO_OPT algorithm when the prefix to be inserted is above the free space pool.

chains in such a way so as to maintain the following invariant. Assume that $D = len(LC(p))$ for a prefix p. Every prefix p is stored in a memory location such that there are at most $\lceil D/2 \rceil$ prefixes between p and a free space entry in the TCAM. Basically, the algorithm distributes the maximal trie chains around the free space pool as equally as possible (see Figure 9).

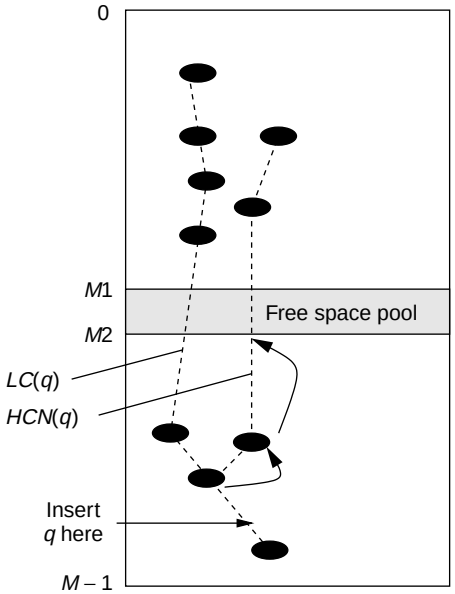

Figure 11. The way an insertion proceeds in the CAO_OPT algorithm when the prefix to be inserted is below the free space pool.

Insertion

Insertion of a new prefix q proceeds in the following manner. First, $LC(q)$ is identified using an auxiliary data structure that's described later. It must be determined whether q needs to be inserted above or below the free space pool (to maintain the balance of $LC(q)$). The two cases are handled separately.

Case I (Figure 10). Assume that q is to be inserted above the free space pool between prefixes p_i and $p_i + 1$ on $LC(q)$. One empty unused entry can be created at that location by moving prefixes on $LC(q)$ downward one by one, starting from p_j to the unused entry at either the top (memory location labeled $m1$ in Figure 10 or the bottom ($m2$) of the free space pool. The total number of movements is clearly less than $D/2$, where D is $len(LC(q))$. The movements don't violate the chain-ancestor ordering constraint since a prefix is moved downward after its ancestor has moved. Hence, the constraint is always satisfied.

Case II (Figure 11). Assume that q is to be

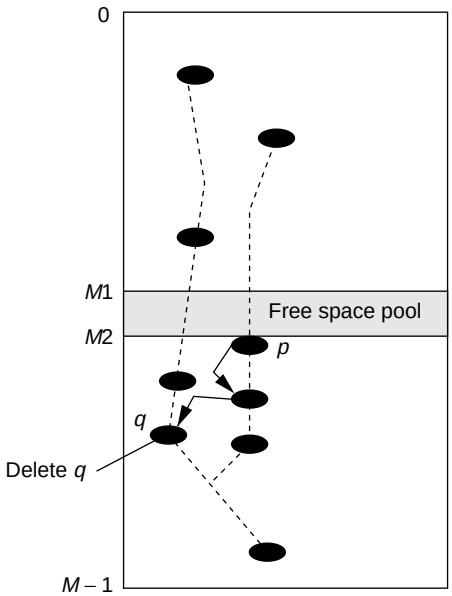

Figure 12. Deletion of a prefix in the CAO_OPT algorithm.

inserted below the free space pool. Creating an empty entry in the TCAM now requires moving the prefixes upward toward the free space pool. Hence, the chain we consider in this case is $HCN(q)$, which may or may not be identical to $LC(q)$. Movement of prefixes one by one upward doesn't violate the chain-ancestor ordering constraint since a prefix is moved to the location previously occupied by the child that occupied the highest memory location among all the children. Again, the total number of movements is clearly less than $D/2$.

Deletion

Deletion is similar to insertion, with the following exceptions:

1. It works in reverse, moving the newly created empty space to the free space pool.
2. It works on the chain that has prefix p adjacent to the free space pool; that is, prefix p is at memory locations $m1 - 1$ or $m2 + 1$.

Figure 12 shows the deletion of a prefix. The new unused entry created by the deletion of prefix q is rippled up by moving prefixes downward on this chain. The total number of movements is less than $D/2$, where D is now either the length of $LC(p)$ if q is deleted from below the free space pool, or the length of $HCN(p)$ if q is deleted from above the free space pool.

Auxiliary trie data structure

The CAO_OPT algorithm maintains an auxiliary trie data structure similar to PLO_OPT to support update operations. However, to determine $LC(p)$ and $HCN(p)$ quickly, we need to keep more information in trie node p. This takes no more than $O(L)$ time by maintaining the following additional fields in every trie node: $wt(p)$, $wt\_ptr(p)$, and $hcld\_ptr(p)$. $wt(p)$ equals

$$\begin{cases} 1 & \text{if } p \text{ is a leaf} \\ \max\left(wt\left(lchild\left(p\right), rchild\left(p\right)\right)\right) & \text{if } p \text{ is not a leaf and not a prefix} \\ 1 + \max\left(wt\left(lchild\left(p\right), rchild\left(p\right)\right)\right) & \text{otherwise} \end{cases}$$

Here, $lchild(p)$ and $rchild(p)$ are the immediate left and right children nodes of p. $wt\_ptr(p)$ keeps a pointer to the prefix child, which has the highest weight, and $hcld\_ptr(p)$ keeps a pointer to the prefix child, which appears at the highest memory location in the TCAM.

Although we haven't proved it, we conjecture that the CAO_OPT algorithm is an optimal online algorithm under the chain-ancestor ordering constraint.

Simulation results

In our simulation we used two publicly available routing-table snapshots (at MAE-EAST and MAE-WEST network access points) and three-hour BGP-update traces on these snapshots taken from Merit (www.merit.edu/ipma/routing_table). Table 1 lists the statistics of the routing tables and BGP updates.

Table 1. Statistics of routing tables and update traces used in simulations.

Type	MAE-EAST	MAE-WEST
Prefixes	43344	35217
Inserts	34204	34114
Deletes	9140	1103

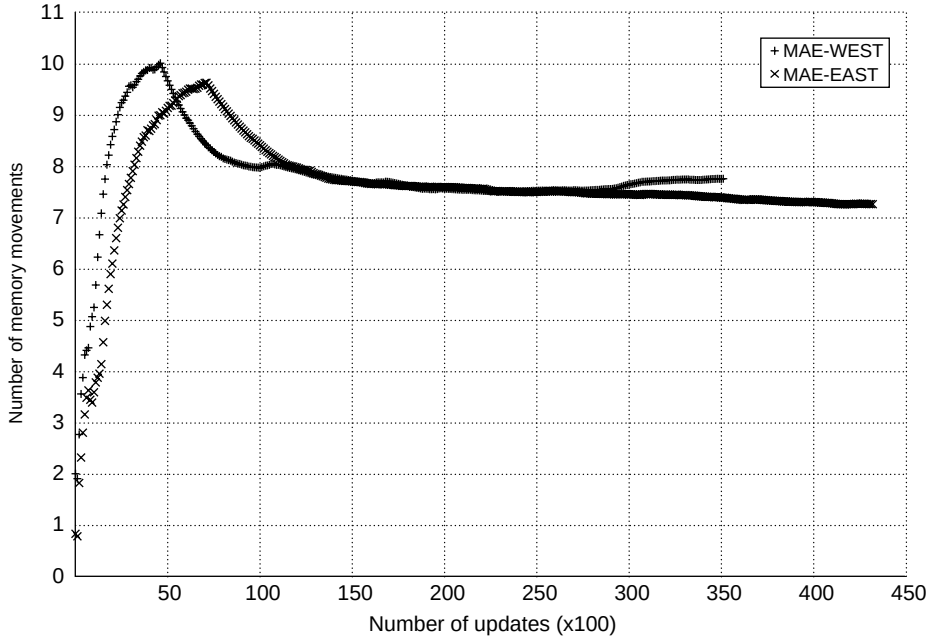

Figure 13. The running average of the number of memory movements required by the *L*-algorithm to support updates on MAE-WEST and MAE-EAST routing tables.

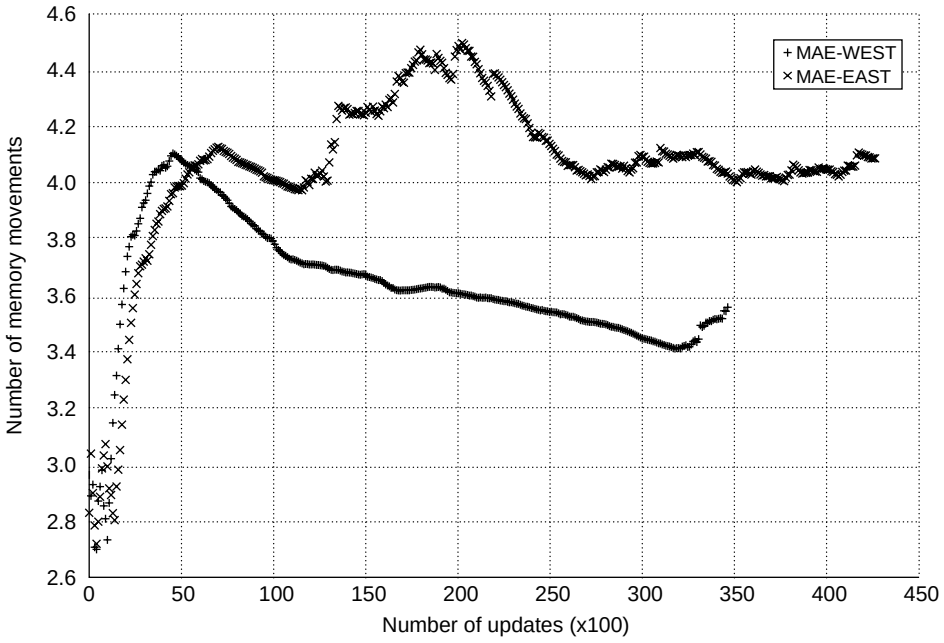

Figure 14. The running average of the number of memory movements required by the PLO_OPT algorithm to support updates on MAE-WEST and MAE-EAST routing tables.

Figure 13 shows a running average of the number of memory movements (memory writes or shifts) required in the *L*-algorithm as a function of the number of updates. The figure shows that the average settles down to around eight memory movements per update operation. This is expected since most of the updates happen to prefixes that are between 8 bits and 24 bits long, because there are very few (less than 0.1%) prefixes that are longer than 24 bits. Hence, if we assume that updates are uniformly distributed between these lengths, the running average should settle at $(24 - 8)/2 = 8$. As shown in Figure 14, the average drops to approximately four memory movements for the PLO_OPT algorithm. This is again expected since theoretical analysis showed an improvement over the *L*-algorithm by a factor of 2.

The motivation for a less stringent constraint (the chain-ancestor ordering constraint) is clear from Figure 15, which plots the maximal chain length distribution of the two routing tables. The figure shows exponentially decreasing distributions; for example, 97% of the MAE-EAST chains have a length less than or equal to two, and all chains have a length less than six.

Figure 16 plots the running average of the number of memory movements required as a function of the number of updates using the CAO_OPT algorithm. This figure shows that the average quickly drops down to 1.02 to 1.06 for both routing tables.

Table 2 and Table 3 (on page 46) list performance summary statistics of both algorithms on the two routing tables. Note that the standard deviation of the CAO_OPT algorithm is

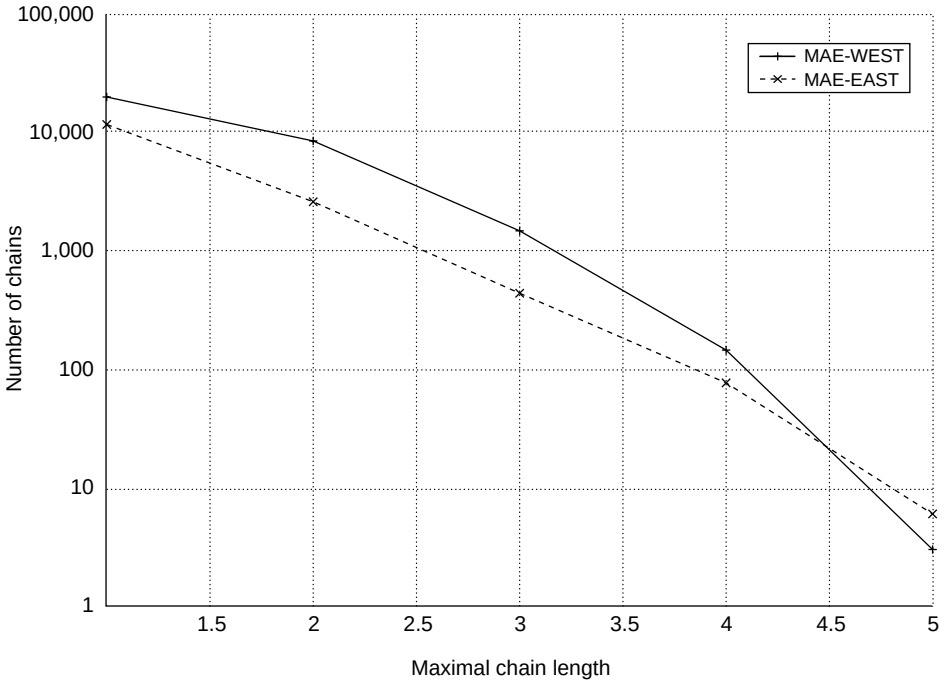

Figure 15. The chain length distribution on the two routing tables. Note the logarithmic scale on the y-axis.

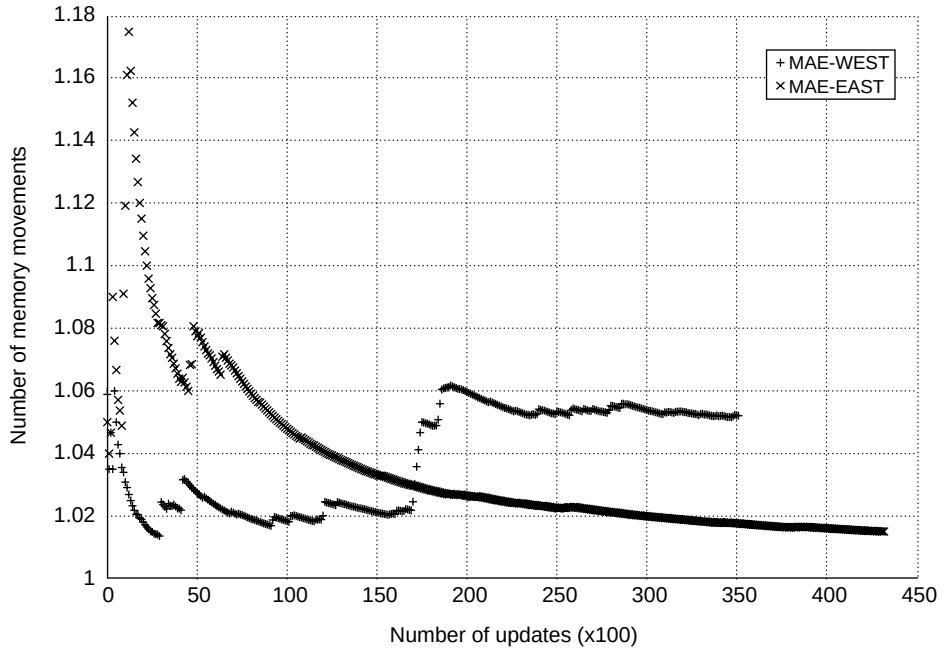

Figure 16. The running average of the number of memory movements required by the CAO_OPT algorithm to support updates on MAE-WEST and MAE-EAST routing tables.

quite small (and much less than that of the PLO_OPT algorithm). This is probably due to the exponentially decreasing chain length distribution. This should make the

Table 2. Summary of performance numbers on MAE-WEST routing table.

Algorithm	Maximum	Average	Standard deviation
L-algorithm	22.0	7.76	3.93
PLO_OPT	13.0	3.56	1.93
CAO_OPT	3.0	1.05	0.02

Table 3. Summary of performance numbers on MAE-EAST routing table.

Algorithm	Maximum	Average	Standard deviation
L-algorithm	21.0	7.27	4.09
PLO_OPT	12.0	4.1	2.03
CAO_OPT	3.0	1.02	0.01

CAO_OPT algorithm even more attractive in practice.

Packet classification

So far, we've discussed updates in the context of routing lookups. Both the PLO_OPT and CAO_OPT algorithms also extend to packet classification.

The prefix-length ordering constraint is equivalent to keeping the list of rules in a classifier ordered by priority. The PLO_OPT algorithm then degenerates to the naive algorithm that requires $O(N)$ memory movements per update in the worst case. Analyzing for the set of overlapping rules and generating a constraint tree could bring this down. Two rules overlap if a packet exists that matches both rules, and only overlapping rules need to be kept in the order of their priority in the TCAM. The constraint tree captures these constraints in a tree form.

Using the constraint tree to determine rule ordering instead of the prefix trie lets us use the CAO_OPT algorithm with little modification. Of course, the benefit of using the chain-ancestor ordering constraint depends on the chain length distribution in the constraint tree, and can only be determined by doing an analysis of real-life classifiers. This task is made difficult by the absence of large publicly available classifiers.

Handling incremental updates in routing lookups can be a slow process—even in simple data structures such as that maintained in a ternary CAM. Both of our proposed algorithms for high-speed updates in TCAMs operate under two separate constraints. Neither requires additional circuitry on the TCAM chip, and one can be proved optimal. In particular, the proposed PLO_OPT algorithm for the stricter (and more well-known) prefix-length ordering constraint improves update speed by a factor of two over the best-known solution.

The CAO_OPT algorithm for the less stringent chain-ancestor ordering constraint guarantees correctness at all times, and completes one prefix update in slightly greater than one (1.02 to 1.06 observed using simulations on real-life routing tables and update traces) memory movement per update operation. Algorithm CAO_OPT is also useful for fast updates when a TCAM is used for packet classification.

The algorithms described here assume that the maximum number of entries is almost the same as the TCAM size. However, we are now interested in whether we can provide better bounds if we're guaranteed that the TCAM won't be occupied more than $(1/k)$th fraction of its size. If we can provide better bounds, can we prove optimality? We could simply extend our two algorithms to a case in which the bounds are reduced from $d/2$ to $d/2k$. Then, we can provide optimality under certain assumptions, but not for the general case. Moreover, it'll be interesting to see if we can achieve an optimal average case algorithm under certain updating distribution. MICRO

Acknowledgments

We gratefully acknowledge Spencer Greene, now at Juniper Networks, for mentioning the possibility of a better algorithm with a less stringent constraint than the prefix-length ordering constraint.

References

1. Y. Rekhter and T. Li, "An Architecture for IP Address Allocation with CIDR," RFC 1518, 1993; http://rfc.net/rfc1518.html.
2. M. Waldvogel et al., "Scalable High-Speed IP Routing Lookups," *Proc. ACM Sigcomm*, ACM, N.Y., 1997, pp. 25–36.
3. A. Brodnik et al., "Small Forwarding Tables for Fast Routing Lookups," *Proc. ACM Sigcomm*, ACM, N.Y., 1997, pp. 3–13.
4. P. Gupta, S. Lin, and N. McKeown, "Rout-

ing Lookups in Hardware at Memory Access Speeds," *Proc. INFOCOM*, IEEE Press, Piscataway, N.J., 1998, pp. 1240–1247.
5. B. Lampson, V. Srinivasan, and G. Varghese, "IP Lookups Using Multiway and Multicolumn Search," *Proc. INFOCOM*, 1998, pp. 1248–1256.
6. S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," *IEEE J. Selected Areas in Communications*, vol. 17, no. 6, 1999, pp. 1083–1092.
7. 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.
8. T.V. Lakshman and D. Stiliadis, "High-Speed Policy-Based Packet Forwarding Using Efficient Multidimensional Range Matching," *Proc. ACM Sigcomm*, ACM, N.Y., 1998, pp. 191–202.
9. V. Srinivasan et al., "Scalable Level 4 Switching and Fast Firewall Processing," *Proc. ACM Sigcomm*, 1998, pp. 203–214.
10. P. Gupta and N. McKeown, "Classifying Packets Using Hierarchical Intelligent Cuttings," *IEEE Micro*, vol. 20, no. 1, Jan. 2000, pp. 34-41.
11. P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. ACM Sigcomm*, ACM, N.Y., 1999, pp. 147–160.
12. M.M. Buddhikot, S. Suri, and M. Waldvogel, "Space Decomposition Techniques for Fast Layer-4 Switching," *Protocols for High Speed Networks*, vol. 66, no. 6, IFIP, Laxenburg, Austria/IEEE Computer Soc., Los Alamitos, Calif., Aug. 1999, pp. 277–283.
13. V. Srinivasan, G. Varghese, and S. Suri, "Fast Packet Classification Using Tuple Space Search," *Proc. ACM Sigcomm*, ACM, N.Y., 1999, pp. 135–46.
14. M. Kobayashi, T. Murase, and A. Kuriyama, "A Longest Prefix Match Search Engine for Multigigabit IP Processing," *Proc. Int'l Conf. on Communications (ICC 2000)*, IEEE Press, Piscataway, N.J., 2000, p. 1360.
15. C. Labovitz, G. R. Malan, and F. Jahanian, "Internet Routing Instability," *IEEE/ACM Trans. Networking*, vol. 6, no. 5, Oct. 1999, pp. 515–28.

Devavrat Shah is working toward his PhD degree in the Computer Science Department at Stanford University. His interests include algorithms for networks, probabilistic analysis of algorithms and queuing theory. Shah received a BTech degree in computer science from IIT-Bombay.

Pankaj Gupta is a PhD candidate in the Computer Science Department, Stanford University. His interests include packet classification and routing lookup for high-speed networks. Gupta received a BTech degree from IIT-Delhi and a PhD degree from Stanford University, both in computer science.

Send questions and comments about this article to Devavrat Shah, 268 Packard-EE Bldg., 350 Serra Mall, Stanford University, Stanford, CA 94305; devavrat@stanford.edu.