Fast incremental updates on Ternary-CAMs for routing lookups and packet classification

Devavrat Shah   *Pankaj Gupta*
Department of Computer Science
Stanford University, CA 94305.
{devavrat,pankaj}@stanford.edu

Abstract— **One popular hardware device for performing fast routing lookups and packet classification is a ternary content-addressable memory (TCAM). A TCAM searches the header of the incoming packet against all entries in the forwarding table or the classifier database in parallel. It keeps the entries in decreasing order of priority of the rules in a classifier, or prefix lengths of the entries in a forwarding table. Keeping the list sorted under addition and deletion of rules in the classifier is an expensive operation, and may take $O(N)$ memory shift (write) operations in the worst case, where N is the number of rules in the classifier (or prefixes in the forwarding table). The most common solutions for this problem improve average case, but waste precious TCAM space, and may still run into the worst case. This paper proposes two algorithms to manage the TCAM such that incremental update times remain small in the worst case. Analysis of these algorithms proves the optimality of one, and suggests that of the other, under the respectively imposed constraints. Finally, simulation results on real data from the Internet shows the performance benefits achievable using these algorithms.**

Keywords—**Routing lookups, packet classification, longest prefix matching, optimality, online algorithms.**

I. Introduction

Internet routers lookup 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 is called the *routing lookup* operation, and is performed on each arriving packet by every router in the path that the packet takes from its source to the destination. The adoption of classless inter-domain routing (CIDR) [1] since 1993 means that a routing lookup needs to perform a "longest prefix match" operation. A router maintains a set of destination address prefixes in a forwarding table. Given a packet, the "longest prefix match" operation consists of finding the longest prefix in the forwarding table that matches the first few bits of the destination address of the packet.

In order to provide enhanced services — such as packet filtering, traffic shaping, policy-based routing etc. — routers also need to be able to recognize *flows*. A flow is a set of packets that obey some *rule*, also called as a *policy*, on the header fields of the packet. These fields include source and destination IP 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 fields of the packet header. The header of an arriving packet 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.

Improvements in optical communication technologies such as DWDM (dense wavelength-division multiplexing) have resulted in continually increasing link speeds — up to 40Gbps per installed fiber at the time of writing. However, routers have been largely unable to keep up at the same pace — a maximum of 10Gbps (OC192) ports are available at the time of writing. 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, [2][3][4][5][6][7] for solutions to the routing lookup problem and [8][9][10][11][12][13] for solutions to the packet classification problem. Many of these papers have indicated the difficulty of the general multi-dimensional packet classification problem in the worst case.

Hardware realizations of algorithmically simpler solutions such as linear search or fully associative search have found favor in some commercial deployments. A popular device is a 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 fully parallel search of the forwarding table or classifier database. The ternary capability allows the TCAM to store wildcards and variable length prefixes by storing don't cares. Lookups are perfomed 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 entries in the TCAM can be intentionally left lying unused in anticipation of future additions — but this leads to wasted space and under-utilization of the TCAM memory. Besides, the worst case still remains $O(N)$.

This paper is motivated by the desire to simultaneously achieve fast incremental updates as well as full utilization of the TCAM. With this objective, the paper describes worst-case algorithms (one specific for route lookups, and the other suitable for both lookups and classification) that achieve the optimal number of TCAM memory 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, the paper shows 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; i.e., 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 [14], and discussed later in the paper.

It turns out that it is not necessary to keep all the forwarding table entries in 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 — though not proved, this algorithm seems to be optimal in the number of worst-case memory operations required to handle a forwarding table update. It is also mentioned how this algorithm and results for routing lookups extend to packet classification.

To the best of our knowledge, there is no previous work that attempts to (algorithmically) optimize updates on a TCAM. Most TCAM vendors live with a $O(N)$ worst-case update time solution. Some attempt to provide a hardware "max" 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 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 at the time of writing this paper). This is likely to get worse in the future as TCAMs scale to greater densities. Another recent paper [15] uses circuit-level optimizations for fast updates at the cost of slower search time and lower memory density.

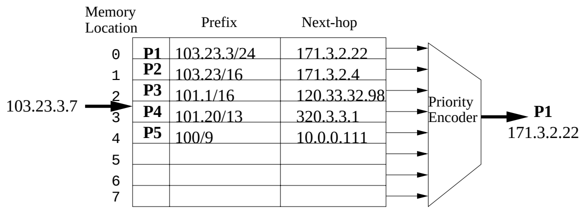

Figure 1. *Longest prefix matching using TCAM*

It should be noted that while the algorithms in this paper 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.

II. Longest-prefix matching using TCAMs

IP addresses are written in the dotted 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 wildcards — this denotes the aggregation of several 32-bit destination IP addresses. For example, the aggregation of 256 addresses 103.23.3.0 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 wildcards. Other examples of prefixes are 101/8, 54.128/10, 38.23.32/21, 200.3.41.1/32 etc. *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, i.e., the most specific, of all the prefixes matching the first few bits of the incoming packet's destination address; and 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

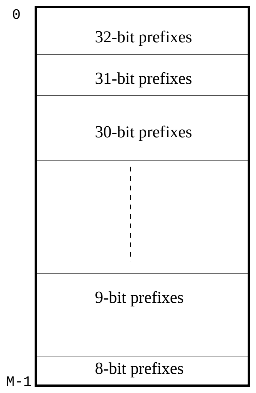

Figure 2. *General configuration of a TCAM used for longest prefix matching. No prefixes of length less than 8-bits are shown, because they are typically not found in forwarding tables*

lengths. The TCAM searches the destination address of an incoming packet with all the prefixes in parallel. Several prefixes (up to L=32 in case of IPv4 lookups) may match the destination address. A priority encoder logic then selects the first matching entry, i.e., the entry with the matching prefix at the lowest physical memory address. An example is shown in Figure 1. The general configuration for storing N prefixes in a TCAM with M memory locations, is shown in Figure 2. We will refer to the set of all prefixes of length j as $\mathcal{P}_j$. We will 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 100-1000 prefixes per second [16]. While this is slow in comparison to the packet lookup rate (which is of the order of millions of packets per second), it is desirable to obtain quick TCAM updates. Slow updates may cause incoming packets to be buffered while an update operation is being carried out, which is undesirable for many applications because it may cause head-of-line blocking and requirement of a large buffer space separate from the main packet buffer memory in the router. Indeed, a single cycle update time is being used by some TCAM vendors [17] as a big competitive advantage. Hence, it is 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 of 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

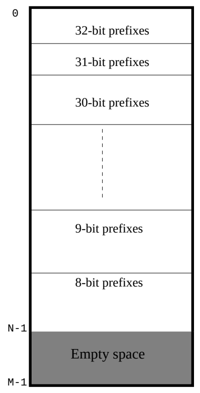

Figure 3. *This naive solution keeps the free space pool at the bottom of memory.*

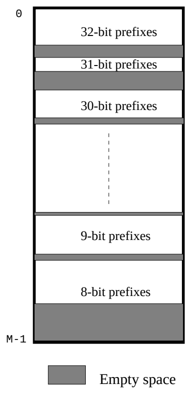

Figure 4. *This solution improves the average case update time by keeping empty spaces interspersed with prefixes in the TCAM.*

to maintain the sorted order. There is a problem since there is no empty space at that location. There can be several ways to handle this issue:

The TCAM manager keeps the free space pool (containing all unutilized 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 downwards in memory by one location each, thus creating an empty space between P1 and P2 where the new prefix can be stored. 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 $N = 64000$, it will take 1.2 milliseconds (assuming one memory write operation can be performed in a 20ns clock cycle) to complete one update operation — too slow for a lookup engine which completes one lookup in 20ns, as a large packet buffer will 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 every X non-empty memory locations, as shown in Figure 4. The

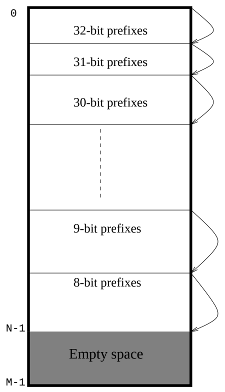

Figure 5. *The prefix-length ordering constraint enables an empty memory location to be found in at most $L = 32$ memory shifts.*

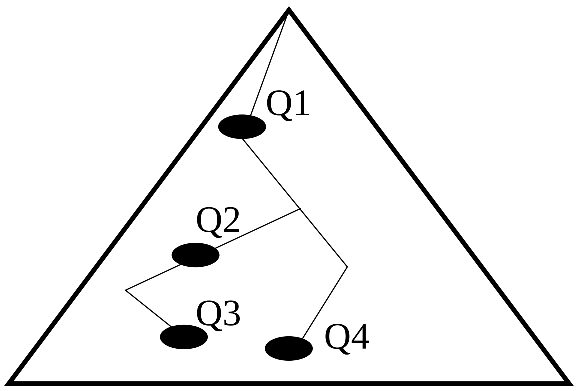

Figure 6. *This figure illustrates the chain-ancestor ordering constraint. There are two maximal chains in this trie: one comprises Q1, Q2 and Q3; and the other comprises Q1 and Q4.*

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 memory space.

The following solution is based on the observation that two prefixes that are of the same length do not need to be in any specific order. This means that if $j > k$, all prefixes in the set $\mathcal{P}_j$ must appear before those in the set $\mathcal{P}_k$, but prefixes within the set $\mathcal{P}_j$ may appear in any order. Hence, there is 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. The average case can be improved again by keeping some empty spaces in between, and not all at the bottom of the TCAM. Section III-A proposes an optimal algorithm, *PLO_OPT*, that brings down the worst case number of memory operations per update to $L/2$.

It turns out that the prefix-length ordering constraint is also more restrictive than what is required for 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 anywhere in the TCAM with respect to prefixes P3, P4 and P5. This is because P1 does not overlap with prefix P3 or P4 or P5 — i.e., no incoming destination address can match both P1 and P3, or P1 and P4, or P1 and P5. Hence, the constraint on ordering of 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 is 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 (i.e., path from the root to a leaf node) of the trie need to be ordered. 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 must be stored at a lower memory location than Q1. We will refer to this constraint as the *chain-ancestor ordering constraint*. Section III-B proposes an algorithm, *CAO_OPT*, that exploits this relaxed constraint to decrease the worst case number of memory operations per update to $D/2$, where D is the maximum length of any chain in the trie. As observed in this section, 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.

III. ALGORITHMS

A. *Algorithm (PLO_OPT) for prefix-length ordering constraint*

The basic idea of algorithm PLO_OPT 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 prefixes of length $L/2-1, L/2-2, ..., 1$ are always below the free space pool. Addition of a new prefix will now have to swap at most $L/2$ memory entries in order to obtain an unused memory entry. Deletion of a prefix is exactly the reverse of addition, and moves the newly created space back to the center of the TCAM. The algorithm keeps a trie data structure in order to do book-keeping of the prefixes stored in the TCAM to support the update operations.

The average case update time can be again improved to better than $L/2$ by keeping some unused entries near each

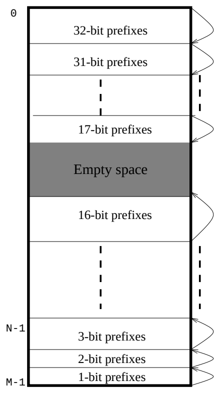

Figure 7. *This figure shows the PLO_OPT algorithm that keeps all the unused TCAM entries in the center of the TCAM such that all prefixes longer than 16-bits are above the empty space, and all prefixes shorter than 16-bits are below the empty space at all times.*

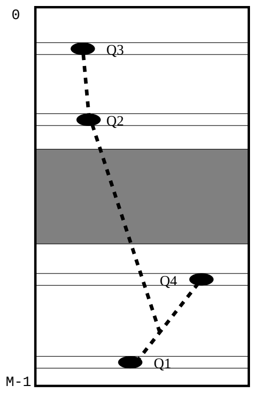

Figure 8. *This figure illustrates the memory assignment of prefixes of Figure 6 under the chain-ancestor ordering constraint. Also shown is the logical inverted trie.*

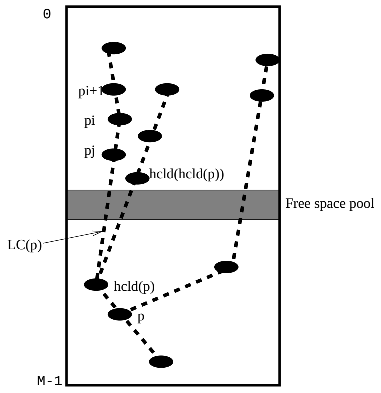

Figure 9. *This figure illustrates the distribution of chains in the TCAM under the chain-ancestor ordering constraint. Every prefix, p, is at a distance less than or equal to $\lceil D/2 \rceil$ prefixes from the free space pool, where $D = len(LC(p))$.*

set $\mathcal{P}_i$, as was done in Figure 4. The worst case number of memory operations is now at least $L/2$, and can become even higher. The distribution of the number of unused entries to be kept around $\mathcal{P}_i$ depends on the distribution of updates, and is therefore difficult to determine apriori. Possible heuristics for placement of empty space include a uniform distribution, or a distribution learned from recently observed update requests.

Algorithm PLO_OPT 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 algorithm PLO_OPT under the prefix-length ordering constraint.

B. Algorithm (CAO_OPT) for chain-ancestor ordering constraint

Before we describe algorithm CAO_OPT, we need some terminology:

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

Algorithm CAO_OPT 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 dotted lines. The basic idea is to arrange the 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).

B.1 Insertion

Insertion of a new prefix q proceeds in the following manner. First, $LC(q)$ is identified using an auxiliary data structure that is described below, and it is determined whether q needs to be inserted above or below the free space pool (to maintain the balance of $LC(q)$). The two

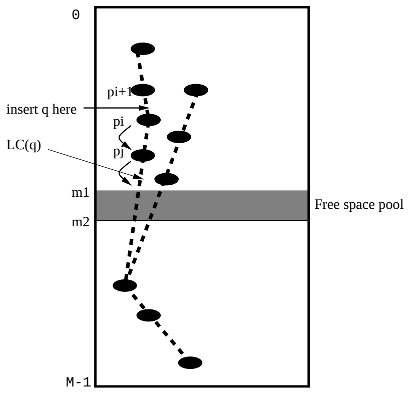

Figure 10. *Showing how insertion proceeds in Algorithm CAO_OPT when the prefix to be inserted is above the free space pool*

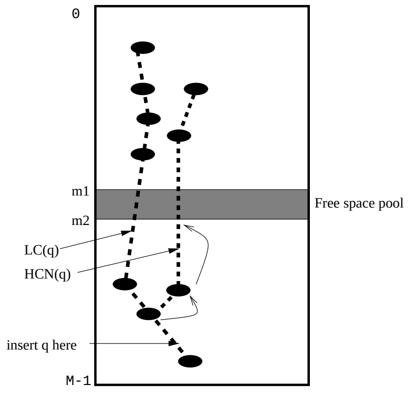

Figure 11. *Showing how insertion proceeds in Algorithm CAO_OPT when the prefix to be inserted is below the free space pool*

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 $LC(q)$ downwards one by one starting from p_j to the unused entry at either the top (memory location marked $m1$ in Figure 10) or the bottom ($m2$) of the free space pool. The total number of movements is clearly less than $\lceil D/2 \rceil$, where $D = len(LC(q))$. The movements do not violate the chain-ancestor ordering constraint since a prefix is moved downwards after its ancestor has moved, and hence, the constraint is always satisfied.

Case II (Figure 11): Now assume that q is to be inserted below the free space pool. Creating an empty entry in the TCAM now requires moving the prefixes upwards towards 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 upwards does not 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 $\lceil D/2 \rceil$.

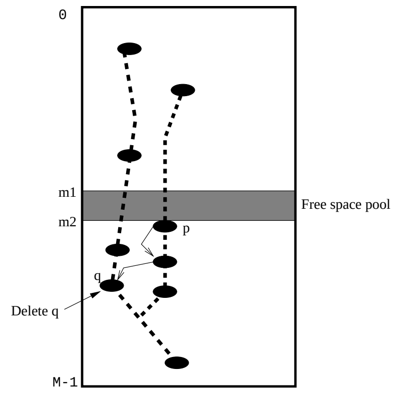

Figure 12. *Deletion of a prefix in Algorithm CAO_OPT*

B.2 Deletion

Deletion is similar to insertion, except: (1) It works in reverse, moving the newly created empty space to the free space pool, and (2) It works on the chain that has the prefix p adjacent to the free space pool — i.e., prefix p is at memory locations $m1 - 1$ or $m2 + 1$. This is shown in Figure 12. The new unused entry created by deletion of prefix q is rippled up by moving prefixes downwards on this chain. The total number of movements is less than $\lceil D/2 \rceil$, 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.

B.3 Auxiliary trie data structure

Algorithm CAO_OPT maintains an auxiliary trie data structure similar to PLO_OPT for supporting the update operations. However, more information is needed to be kept in a trie node,p, to determine $LC(p)$ and $HCN(p)$ quickly. 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:

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

where $lchild(p)$ and $rchild(p)$ are the immediate left and right children nodes of p respectively. $wt\_ptr(p)$ 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.

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

TABLE I
STATISTICS OF ROUTING TABLES AND UPDATE TRACES USED IN THE SIMULATIONS.

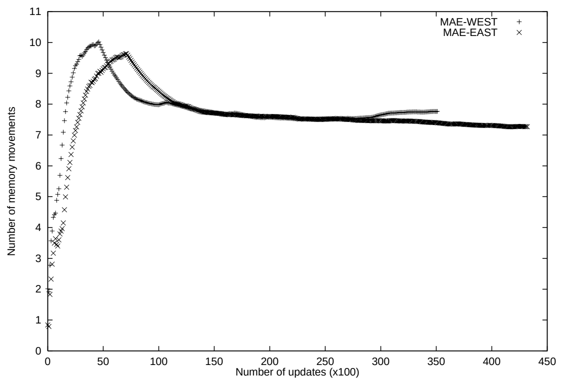

Figure 13. *The running average of the number of memory movements required by the L-algorithm to support updates on MAE-WEST and MAE-EAST routing tables.*

Though we have not been able to prove, we conjecture that algorithm CAO_OPT is an *optimal* online algorithm under the chain-ancestor ordering constraint.

IV. SIMULATION RESULTS

This section presents simulation results using 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 [18]. Statistics of the routing tables and BGP updates are shown in Table I.

Figure 13 shows a running average of the number of memory movements (i.e., 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 8 memory movements per update operation. This is expected since most of the updates happen to prefixes that are between 8 and 24 bits long, because there are very few prefixes (less than 0.1%) 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 4 for algorithm PLO_OPT. 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-

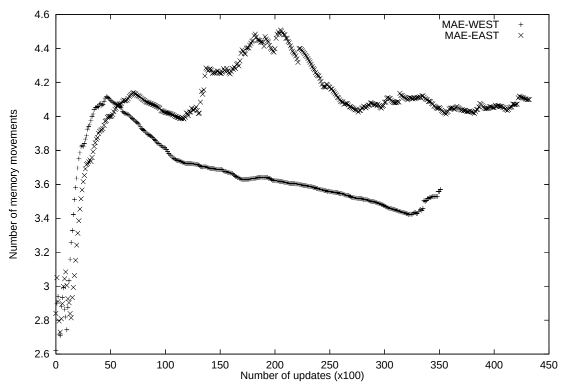

Figure 14. *The running average of the number of memory movements required by algorithm PLO_OPT to support updates on MAE-WEST and MAE-EAST routing tables.*

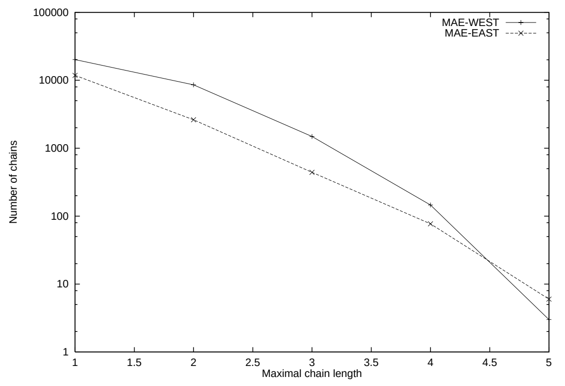

Figure 15. *The chain length distribution on the two routing tables. Note the logarithmic scale on the y-axis*

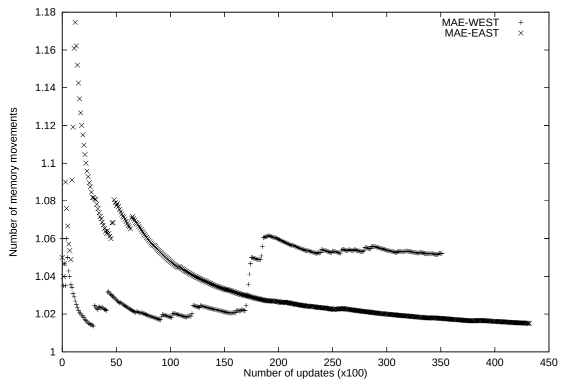

Figure 16. *The running average of the number of memory movements required by algorithm CAO_OPT to support updates on MAE-WEST and MAE-EAST routing tables.*

	Max	*Avg*	*Std Dev*
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 II
SUMMARY OF PERFORMANCE NUMBERS ON MAE-WEST ROUTING TABLE.

	Max	*Avg*	*Std Dev*
L-algorithm	21.0	7.27	4.09
PLO_OPT	12.0	4.1	2.03
CAO_OPT	3.0	1.02	0.01

TABLE III
SUMMARY OF PERFORMANCE NUMBERS ON MAE-EAST ROUTING TABLE.

ancestor ordering constraint) is immediate 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 length less than or equal to two and all chains have 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 algorithm CAO_OPT. This figure shows that the average drops quickly down to 1.02-1.06 for both routing tables.

Performance summary statistics of both algorithms on the two routing tables is shown in Table II and Table III. It is to be noted that the standard deviation of algorithm CAO_OPT is quite small (and much less than that of algorithm PLO_OPT) — probably because of the exponentially decreasing chain length distribution. This should make algorithm CAO_OPT even more attractive in practice.

V. PACKET CLASSIFICATION

So far we have discussed updates in the context of routing lookups. Both algorithms PLO_OPT and CAO_OPT extend to packet classification as follows.

The prefix-length ordering constraint is equivalent to keeping the list of rules in a classifier ordered by priority. Algorithm PLO_OPT then degenerates to the naive algorithm that requires $O(N)$ memory movements per update in the worst case. This could be brought down by doing an analysis to determine the set of overlapping rules and generating a *constraint tree*. Two rules overlap if there exists a packet which matches both rules. 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. Now, algorithm CAO_OPT can be used with little modification, using the constraint tree to determine rule ordering instead of the prefix trie. Of course, the benefit of using the chain-ancestor ordering constraint is dependent on the chain length distribution in the constraint tree, and can only be determined by doing an analysis of real-life classifiers — a task made difficult by the absence of large publicly available classifiers.

VI. CONCLUSIONS

Handling incremental updates in routing lookups can be slow — even in simple data structures such as that maintained in a ternary CAM. This paper proposes algorithms for high speed updates in TCAMs under two separate constraints. Both algorithms do not require any additional circuitry on the TCAM chip and one can be proved optimal. In particular, the paper proposes algorithm PLO_OPT for the stricter (and more well-known) prefix-length ordering constraint, and achieves a factor of two update speed improvement over the best known solution. This paper also proposes algorithm CAO_OPT for the less stringent chain-ancestor ordering constraint. Algorithm CAO_OPT guarantees correctness at all times, and completes one prefix update in slightly greater than one (observed 1.02-1.06 using simulations on real-life routing tables and update traces) memory movements per update operation. Algorithm CAO_OPT is also useful for fast updates when a TCAM is used for packet classification.

VII. ACKNOWLEDGMENTS

We would like to 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.

[2] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high-speed ip routing lookups," in *Proceedings of ACM SIGCOMM*, Oct. 1997, pp. 25–36.

[3] A. Brodnik, S. Carlsson, M. Degermark, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proceedings of ACM SIGCOMM*, Oct. 1997, pp. 3–13.

[4] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proceedings of INFOCOM*, Mar. 1998, pp. 1240–7.

[5] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," in *Proceedings of INFOCOM*, Mar. 1998, pp. 1248–1256.

[6] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083–92, 1999.

[7] V. Srinivasan and G. Varghese, “Fast address lookups using controlled prefix expansion,” *ACM Transactions on Computer Systems*, vol. 17, no. 1, pp. 1–40, Oct. 1999.

[8] T. V. Lakshman and D. Stiliadis, “High-speed policy-based packet forwarding using efficient multi-dimensional range matching,” in *Proceedings of ACM SIGCOMM*, Sept. 1998, pp. 191–202.

[9] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, “Scalable level 4 switching and fast firewall processing,” in *Proceedings of ACM SIGCOMM*, Sept. 1998, pp. 203–214.

[10] P. Gupta and N. McKeown, “Classifying packets using hierarchical intelligent cuttings,” *IEEE Micro*, vol. 20, no. 1, pp. 34–41, Jan-Feb 2000.

[11] P. Gupta and N. McKeown, “Packet classification on multiple fields,” in *Proceedings of ACM SIGCOMM*, Sept. 1999, pp. 147–60.

[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, pp. 277–83, Aug. 1999.

[13] V. Srinivasan, G. Varghese, and S. Suri, “Fast packet classification using tuple space search,” in *Proceedings of ACM SIGCOMM*, Sept. 1999, pp. 135–46.

[14] “Sibercore technologies,” www.sibercore.com/scan01_cidr_p03.pdf.

[15] M. Kobayashi, T. Murase, and A. Kuriyama, “A longest prefix match search engine for multi-gigabit ip processing,” in *Proceedings of ICC 2000*, June 2000.

[16] C. Labovitz, G. R. Malan, and F. Jahanian, “Internet routing instability,” *The IEEE/ACM Transactions on Networking*, vol. 6, no. 5, pp. 515–28, Oct. 1999.

[17] “Netlogic microsystems,” www.netlogicmicro.com.

[18] “Merit,” www.merit.edu/ipma/routing_table.