SORTING AND SEARCHING USING TERNARY CAMS

SORTING AND SEARCHING ARE CLASSIC PROBLEMS IN COMPUTING. ALTHOUGH SEVERAL RAM-BASED SOLUTIONS EXIST, ALGORITHMS USING TERNARY CONTENT-ADDRESSABLE MEMORIES OFFER PERFORMANCE BENEFITS. USING THESE ALGORITHMS, A TCAM CAN SORT IN $O(N)$ MEMORY CYCLES.

Rina Panigrahy
Samar Sharma
Cisco Systems

• • • • • • As the computing world evolves, the need for faster, feature-rich, and cost-effective solutions increases. To obtain higher performance, designers are building more features in hardware using application-specific integrated circuit and field-programmable gate array technology.

Many applications encounter sorting and range search problems. For example, in layer 4 switching, you might search for a port number in a set of port ranges. Range search is also a commonly performed operation in packet classification.[1-4]

Another example of sorting and range search problems occurs in a distributed-storage networking scenario where many hosts might access shared data at very high speeds. The hosts protect their access by locking the accessed address range. A device might check whether another device is already accessing a specific location before beginning a memory access.

More generally, before locking a range, a device might check for whether any address in the range overlaps with ranges that other devices are accessing. The set of ranges changes dynamically, as devices add new locks and release old ones.

In fact, range search is a key component of the general locking problem. Other high-speed locking applications occur in memory protection, Small Computer System Interface (SCSI) Reserve in multihost environments, memory accesses between processes, and accesses to a shared memory in multiprocessor environments. Range search is also useful in memory management to ensure that processes access only the allocated memory ranges.

We studied two types of range search problems:

- The point intersection problem determines whether a set of ranges contains a query point. The application might dynamically insert or delete ranges from the set.
- The range intersection problem determines whether a query range intersects with any range in a set of ranges.

These are classic problems in computing, and several RAM-based solutions exist.[2,5-7]

Here, we present several ternary content-addressable memory (TCAM)-based algorithms for the dynamic point and range intersection problems, both for disjoint and nondisjoint ranges.

A TCAM stores data with three logic values: 0, 1, or X (don't care). Each TCAM entry contains a value and a mask. The entries are stored in the TCAM in decreasing order of priority. The TCAM compares a given input against all the entries in parallel, returning the first entry that matches the input. An entry matches the input if the input and the entry values are identical in the unmasked bits.

Initially, routers and switches primarily used TCAMs to perform forwarding lookups for Internet Protocol addresses. TCAMs can also easily perform longest-prefix matching. Currently, networking products use TCAMs for packet classification, network address translation, route lookups in storage networks, layer 4 to layer 7 switching, server load balancing, label switching, and high-performance firewall functions. Because several networking products already use TCAMs, designers could easily leverage the TCAMs in these products to solve other problems.

Using our algorithms, you can efficiently perform sorting and priority queue operations. We provide a scheme to perform inserts and deletes to a sorted list in $O(1)$ time. Searching and maintaining a sorted list of numbers is a common operation in many high-speed applications.

Point and range intersection problems

In this section, we formally define the point intersection and range intersection problems. Let S be a set of ranges, $S = \{[s_1, t_1], [s_2, t_2], ..., [s_n, t_n]\}$, where s_i and t_i and are integers such that $0 \leq s_i \leq t_i \leq W, \forall i$. W is a bound on the largest endpoint of the ranges.

Without loss of generality, we assume the ranges are inclusive. That is, range $[s_i, t_i]$ includes s_i and t_i.

We summarize the point intersection problem as follows: Given integer x, $0 \leq x \leq W$, determine whether any range in S contains x. We call this a *search* operation. We also need to support the dynamic insertion and deletion of ranges to the set.

The point intersection problem has two flavors. In the first case, the ranges in S are disjoint—they do not overlap. We call this case the point intersection problem for disjoint ranges. Figure 1 shows an example of point intersection for three disjoint ranges. In the general case, the ranges in S may not be disjoint.

We define the range intersection problem as follows: Given a query range, determine whether any range in S intersects with x. Again, set S can change dynamically.

Related work

Researchers have proposed several RAM-based solutions for the range search problem.[5,6] In general, the data structures are very efficient but not easily realizable in hardware. The best-known theoretical result for solving the static point intersection problem takes $O[\log(\log W)]$ memory accesses using space $O(n)$ with a large preprocessing time, and the constants involved are of moderate size.

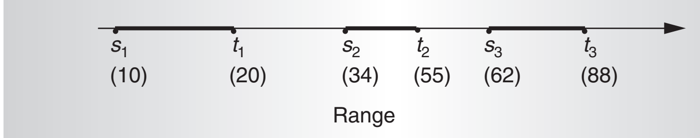

Figure 1. Example of point intersection for disjoint ranges.

The best-known theoretical result for sorting using RAM requires $O[n\log(\log n)]$ operations; however, we acknowledge that this is only a theoretical result and unsuitable for practical implementation.[8]

Our goal here is to provide efficient algorithms for TCAMs that are easily implementable in hardware. Furthermore, the constants in our algorithms' time and space complexity are smaller than those of most RAM-based solutions.

One scheme to solve the point intersection problem for disjoint ranges using a TCAM is to break each range into at most $2\log W$ ranges (http://www.sibercore.com/products_AppNotes.htm).[4] This enables search in one lookup but uses up to $2\log W$ TCAM entries per range, and uses $O(\log W)$ TCAM operations for insertions and deletions.

For the disjoint case, we provide algorithms that use two prefix entries per range and perform search, insertion, and deletion in two TCAM operations. Furthermore, our algorithms do not require maintaining tree data structures.

Our results

We provide different algorithms for the case when the ranges are disjoint and for the general case.

For disjoint ranges, we first present results for algorithm PIDR_OPT in Table 1; it solves the point intersection problem. PIDR_OPT uses only two TCAM lookups to perform a search operation, two TCAM operations for a range insertion operation, and two TCAM entries per range in the set.

An enhancement of this algorithm yields algo-

Table 1. Point intersection for disjoint ranges.

Algorithm	No. of TCAM lookups per search	No. of operations per insertion or deletion	No. of TCAM entries per range	TCAM requirement
PIDR_OPT	2	2	2	Assumes longest-prefix matches
PIDR2	2log(log *W*)	2	2	Double-width TCAM entry
PIDR3	log *W*	1	1	None

Table 2. Range Intersection for disjoint ranges.

Algorithm	No. of TCAM lookups per search	No. of operations per insertion or deletion	No. of TCAM entries per range	TCAM requirement
RIDR_OPT	2	4	4	Assumes longest-prefix matches
RIDR2	2log(log *W*)	4	4	Double-width TCAM entry
RIDR3	log *W*	2	2	None

rithm RIDR_OPT for the range intersection problem. As Table 2 shows, this algorithm uses two TCAM lookups for search, four TCAM operations for range insert, and four TCAM entries per range. These algorithms assume longest-prefix matches during TCAM lookups.

Ensuring longest-prefix matches during TCAM lookups requires TCAM space management.[9] To avoid this overhead, we present other solutions. Algorithms PIDR2 and RIDR2 eliminate the longest-prefix match requirement by using a TCAM of double the width. However, these algorithms use 2log(log W) TCAM lookups per search operation. The other operations' time and space complexity remain the same.

Even without using a wider TCAM and without assuming longest-prefix lookups, we have a solution (PIDR3) for point search that uses log W TCAM operations for a point search and one TCAM operation for a range insertion. PIDR3 uses only one TCAM entry for each range. In fact, this algorithm does not even use the TCAM's ternary nature and is implementable using a CAM. Again, we provide a similar solution (RIDR3) for the range intersection problem.

The case in which the ranges are not disjoint is more difficult. For the general, dynamic range search problem, prior art requires one TCAM lookup for a search operation and $O(\log W)$ TCAM inserts for an insert operation, and uses $O(\log W)$ TCAM entries per range.[4] We present algorithms PI_OPT and RI_OPT that use constant time for search, constant space per range, and $O(\log W)$ time for insertions and deletions. We summarize the results for these algorithms in Table 3.

These algorithms assume longest-prefix matches. Again, we can eliminate this requirement (in algorithms PI2 and RI2) by using a double-width TCAM and increasing the worst-case lookup time to $O[\log(\log W)]$.

Finally, we provide simple, practical CAM-based solutions PI3 and RI3 that use $O(\log W)$ time for all operations and constant space per input range.

With these algorithms, you can perform sorting in $O(n)$ time. Furthermore, you can perform insertions and deletions to a sorted list and to priority queue operations in $O(1)$ time.

Algorithms for disjoint range problems

A central concept in our algorithms is determining the *longest common prefix* of any two points s and t, denoted by LCP(s, t). Given range $R = (s, t)$, we can compute the LCP, P, for that range by determining the longest common prefix of its endpoints. We say that P is the LCP for range R. Points $P01 \ldots 1$ and $P10 \ldots 0$ must be present in range R. This implies that for a set of disjoint ranges, the LCP for each range is

Table 3. Point and range intersection with possibly nondisjoint ranges.

Algorithm	No. of TCAM lookups per search	No. of operations per insertion or deletion	No. of TCAM entries per range	TCAM requirement
PI_OPT or RI_OPT	4	4log *W*	4	Assumes longest-prefix matches
PI2 or RI2	4log(log *W*)	2log *W*	4	Double-width TCAM entry
PI3 or PI3	log *W*	2log *W*	2	None
Prior art				
PI	1	2log *W*	2log *W*	None
RI	2	3log *W*	3log *W*	None

unique. So we can say that *R* is the range for *P*.

Point intersection

A point *x* matches *P* if *P* is a prefix of *x*. Any query point *x* that belongs to a range must match the LCP for that range.

We must check whether *x* belongs to a range only if *x* matches its LCP. Unfortunately, *x* could match the LCPs for many ranges, and we would need to check this for all such ranges, because the LCPs themselves could be prefixes of each other. This situation poses the following interesting question: Is it sufficient to examine only the longest LCP matching *x*? If so, it would suffice to perform the range check for only the longest matching LCP. However, we can show this statement to be false. Surprisingly, a variant of this statement turns out to be true.

We prove that by appropriately modifying the set of LCPs and dividing it into two classes, you need only look at the longest prefix in each class, as follows. To simplify the presentation, we assume that all ranges are at least of length two. It is easy to handle point ranges seamlessly.

For each LCP *P*, look at prefixes P1 and *P*0 obtained by extending *P* by one bit. We call these prefixes extended LCPs (ELCPs), where *P*1 is a 1-ELCP and *P*0 is a 0-ELCP.

If *x* matches an LCP, it must then match one of its two ELCPs. Further, the ELCPs for range *R* divide the range into two parts. The points on the left part match the 0-ELCP, and the points on the right part match the 1-ELCP.

For each range [*s*, *t*] in the set, we produce the pair of ELCPs, and divide the ELCPs into two classes, placing the 0-ELCPs in class C_0 and the 1-ELCPs in class C_1. In theorem A, we prove that for disjoint ranges, if *x* is in range *R*, then either the 0-ELCP or the 1-ELCP for *R* must be the longest prefix in its class that matches *x*.

```
// ELCP→q stores the left endpoint of its range if
// it is 0-ELCP and the right endpoint if it is 1-ELCP

PIDR_Search (x) {
    ELCP0 = Lookup x in class C0
    if (there exists ELCP0) and (x ≥ ELCP0→q)
        return (ELCP0→range)
    ELCP1 = Lookup x in class C1
    if (there exists ELCP1) and (x ≤ ELCP1→q)
        return (ELCP1→range)
}

insert (s, t) {
    P = longest_common_prefix(s, t);
    store P0* in TCAM class C0 and point it to s.
    store P1* in TCAM class C1 and point it to t.
}

delete (s, t) {
    P = longest_common_prefix(s, t);
    // To locate ELCPs exactly, pad P as follows
    lookup P01…1 in the class C0.
    Delete that entry
    lookup P10…0 in the class C1.
    Delete that entry.
}
```

Figure 2. Algorithm PIDR_OPT.

This gives us the following algorithm for solving the point intersection problem for disjoint ranges. Store the 0-ELCPs for the input ranges in TCAM class C_0 and 1-ELCPs in separate TCAM class C_1. To search for point *x*, we look up the longest matching prefixes in C_0 and C_1, and check whether *x* lies in any of their ranges. To enable this check, we store the left endpoint for its range with each 0-ELCP and the right endpoint of its range with each 1-ELCP.

The algorithm in Figure 2 uses two TCAM

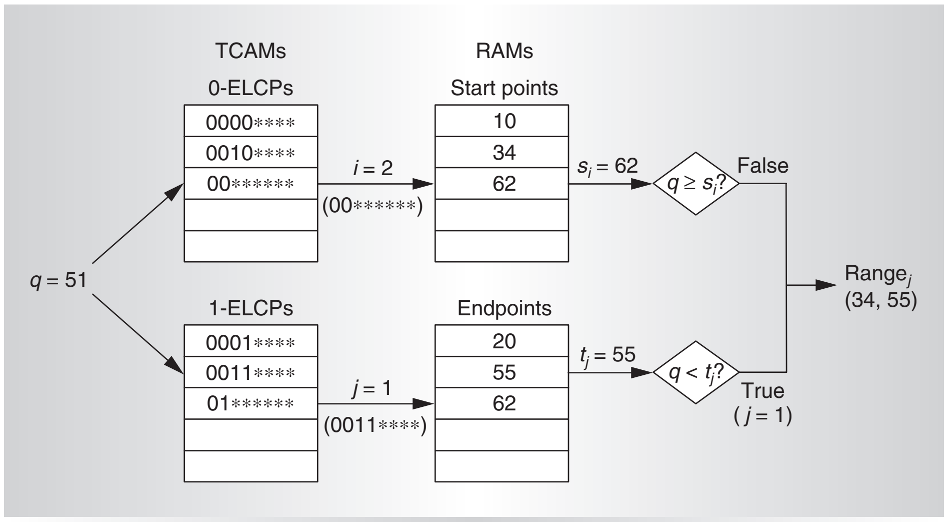

Figure 3. Range search using TCAMs.

lookups for a search operation, two TCAM operations for insert and delete, and two TCAM entries per range.

Theorem A. For a set of disjoint ranges, if x is in range R in the set, then either the 0-ELCP or the 1-ELCP for R must be the longest prefix in its class that matches x.

Proof. Let P denote the LCP for R; x must match one of the ELCPs, $P1*$ or $P0*$ (the asterisk denotes any string of 0s or 1s and indicates that $P1$ and $P0$ are prefixes). Without loss of generality, assume that it matches $P1*$. We must prove that class C_1 doesn't contain longer prefixes that match x.

Let us assume the contrary, saying that x matches a longer prefix in C_1. If so, the prefix must be of the form $P1\,Q1*$, where Q is a binary string.

Let R' denote the range for the ELCP, $P1\ Q1*$. We will prove that ranges R and R' must have at least two points in common, which poses a contradiction. These two points are $u = P1\,Q01 \ldots 1$ and $v = P1\,Q10 \ldots 0$ (We use $0 \ldots 0$ to denote an unspecific number of zeroes and $1 \ldots 1$, an unspecific number of ones.) Because $P1\,Q$ is the LCP for range R', clearly the above points are in R'. To prove that they are in R, we will show that they lie between two points in R.

Consider $y = P01 \ldots 1$. Clearly, u and v are greater than y. Also, because x matches $P1\,Q1*$, $u < x$, and $v \leq x$. So we have : $y < u < v \leq x$.

Clearly y is in R; x is given to be in R. So, u and v must be in R, thus posing a contradiction.

Note that the proof works even if the ranges intersect only at their endpoints. That is, in set $S = \{[s_1, t_1], [s_2, t_2], \ldots, [s_n, t_n]\}$, t_i can equal s_{i+1}.

We will use this fact for solving the range intersection problem.

Example

We now apply our algorithm to the example in Figure 1, treating all numbers as 8-bit quantities. The LCPs of the three ranges turn out to be 000*****, 001*****, and 0*******. The 0-ELCPs are 0000****, 0010****, and 00******. The 1-ELCPs are 0001****, 0011****, and 01******.

Two TCAMs can perform the range search on these ranges, executing longest-prefix matches concurrently among the 0-ELCPs and the 1-ELCPs. Figure 3 illustrates this process.

For example, let's say query point q is 51. The algorithm looks this point up in the two TCAMs. The matching rows are $i = 2$ and $j = 1$. The algorithm then retrieves corresponding endpoints $s_2 = 62$ and $t_1 = 55$ and compares against q. Because the second comparison is successful, jth range (34, 55) is the correct matching range. If both comparisons had been unsuccessful, then the algorithm would not output any range. The two TCAM, static RAM lookups, and the two comparisons can execute concurrently in hardware. More precisely, hardware can perform disjoint range search in one memory cycle using a two-stage pipeline.

Range intersection

We can reduce any range intersection problem (disjoint or nondisjoint set) to two point intersection problems as follows.

$S = \{[s_1, t_1], [s_2, t_2], \ldots, [s_n, t_n]\}$, is the set of ranges for the range intersection problem. Given query range $[x, y]$, first search for point y in S. If there is a match, return that range as an intersecting range.

Otherwise, we look for y in the following set of ranges: $S' = \{[0, t_1], [t_1, t_2], ..., [t_{n-1}, t_n], [t_n, W]\}$.

Say we find y to lie between t_i and t_{i+1}. Then we compare x with t_i. If $x \le t_i$, we return $[s_i, t_i]$ as an intersecting range. Otherwise, there is no intersection.

For disjoint ranges, we can combine the two lookups into one by looking for y in the following set of ranges: $S'' = \{[0, s_1], [s_1, t_1], [t_1, s_2], ..., [s_n, t_n], [t_n, W]\}$. As in the point intersection problem, we store the ELCPs for each range in set S in two TCAM classes.

During an insertion of $[s, t]$ between t_1 and s_{i+1}, it might seem as if we must delete LCPs for $[t_i, s_{i+1}]$ and insert LCPs for $[t_i, s]$, $[s, t]$, and $[t_i, s_{i+1}]$. However, the following lemma shows that we don't need to delete any LCPs and can perform fewer TCAM insertions. (We omit the lemma's proof in this article.)

Lemma. If $x \le y \le z$ then LCP(x, z) is a subprefix of both LCP(x, y) and LCP(y, z) and equals one of those.

P is a subprefix of Q if length$(P) \le$ length(Q), and most-significant length(P) bits are identical in P and Q.

Figure 4 implements the algorithms for the range intersection problem.

```
// For each endpoint s or t, we maintain whether it is a
// start point or an endpoint of the input range and its
// corresponding input range.

RIDR_Search(x, y) {
    [u, v] = PIDR_Search(y)
    if u is a start point in an input range
        return [u, v]
    elseif (y == v)
        return range corresponding to v
    else { // u is an ending point
        if (x ≤ u)
            return range corresponding to u
        else
            return no intersection
    }
}

Point_lookup (p) {
    ELCP0 = Lookup p in class C0
    If (ELCP0 exists) and (p ≥ ELCP0→q)
        then return ELCP0→q
    ELCP1 = Lookup p in class C1
    If (ELCP1 exists) and (p ≤ ELCP1→q)
        then return ELCP1→q
}

Insert_point(p) {
    v = Point_lookup(p)
    P = longest_common_prefix(p,v)
    Insert P0 pointing to smaller of p and v
    Insert P1 pointing to larger of p and v
}

insert (s, t) {
    insert_point(s)
    insert_point(t)
}

delete_point(p) {
    P = longest ELCP matching P obtained by doing
        2 lookups in C0 and C1
    Delete P and its sibling from the TCAM
    // sibling of a prefix is obtained by inverting the last bit
    // of the prefix
}
delete (s, t) {
    delete_point(s)
    delete_point(t)
}
```

Figure 4. Algorithm RIDR_OPT.

Sorting

We can sort n numbers by making n insertions into an empty sorted list. Inserting into a sorted list is equivalent to performing a range search on the ranges that the list elements define. These ranges are disjoint and cover the entire line. If $(a_1, a_2, a_3, ...)$ are elements in sorted order, then the corresponding ranges are $[0, a_1)$, $[a_1, a_2)$, ..., $[a_n, W)$. We maintain a doubly linked list to enable easy insertion in the middle of this list. The insertion algorithm is very similar to the insert_point() routine used in PIDR_OPT. This requires two insertions into the TCAM and three additional memory operations for insertion into the doubly linked list. Using our algorithm, you can perform insertions and deletions to a sorted list in $O(1)$ time. Thus, you can sort n numbers in $O(n)$ time and perform priority queue operations in $O(1)$ time.

Eliminating the longest-prefix match requirement

Algorithms PIDR_OPT and RIDR_OPT require longest-prefix matches in the TCAM. One way to accomplish these matches is by storing the prefixes in decreasing-length order, which involves TCAM space management overhead. You must then maintain this order during insertions. This might require moving certain TCAM entries during an insertion to maintain this order. Researchers have devised various techniques to minimize the moves during an insertion.[9] Clearly having a TCAM of size $n\log W$ completely eliminates all moves.

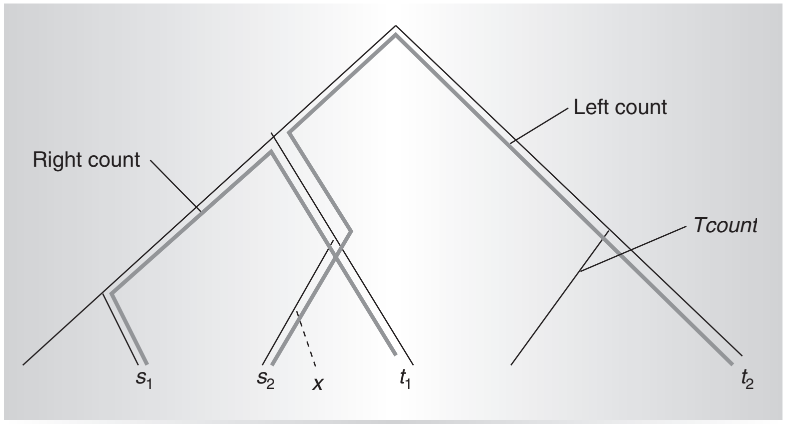

Figure 5. Patricia tree on the endpoints.

We now describe a method to perform longest-prefix match without maintaining prefix order; it uses a double-width TCAM. The scheme uses log(log W) TCAM lookups for the longest-prefix lookup.

We divide the TCAM into two logical columns of width w = log W. For every prefix P, we store $P*^{l-1}1*^{w-l}$ in the TCAM, where l = length(P).

To look up key k, we perform a binary search for the length of the longest prefix that matches k by feeding log(log W) different values into the second part of the TCAM lookup key. The first part of the TCAM lookup key is always set to k.

We search for prefixes of length greater than $w/2$ by feeding $0^{w/2}1^{w/2}$ into the second part of the TCAM lookup key. If we find a match, we look for prefixes with lengths between $3w/4$ and w by feeding $0^{3w/4}1^{w/4}$. Otherwise, we look for prefixes with lengths between $w/4$ and $w/2$ by feeding $0^{w/4}1^{w/4}0^{w/2}$, thus continuing the binary search.

For the point intersection problem, if the ranges are much smaller than W and scattered apart, the LCPs will not likely be subprefixes of each other. This characteristic gives us a technique to prune the binary search space in looking for the longest TCAM entry. Look up key k in the TCAM by feeding 1^w to the search in the second part, and say we find prefix P of length l. You can look for prefixes longer than that by feeding 0^l1^{w-l} in the second part of the TCAM lookup key. If there is no such longer key, then we will learn that immediately. So for such applications, the search would typically take two lookups.

Another practical solution for applications that can afford log W TCAM operations for lookup is to simply try all the key's prefixes. In fact, this solution does not even use the TCAM's ternary nature, and you can accomplish it using an exact lookup in a CAM. Store prefix P as Pb^{w-l} in a CAM, where l = length(P), and b is the bit obtained by inverting the last bit of P. This gives us an algorithm with O(log W) search time, O(1) insertion time, and O(1) CAM space per range for disjoint range intersection and range search problems.

General-purpose algorithms

One method for solving the point intersection problem for possibly overlapping ranges is to break every range into O(log W) prefixes.[4] This enables search in O(1) time and insertions in O(log W) time, and uses O(log W) space per range. This method also yields an algorithm of the same complexity for the range intersection problem by using the reduction described earlier.

We now present an algorithm for the general point intersection problem; it uses O(1) time for search and O(log W) time for insertion, and uses O(1) space per range. We obtained these results by exploiting the fundamental similarity between a Patricia tree and LCPs. A Patricia tree is a binary tree where bits of a key are used to walk down from the root, a left child is taken on a 0 bit; and a right child, on a 1 bit. For sorted set of points (a_1, a_2, a_3, ...), the set of LCPs of adjacent points actually represents the Patricia tree on these points. These LCPs are in fact exactly all of the internal nodes in the Patricia tree. Our algorithms demonstrate simple methods of exploiting the Patricia tree data structure (without explicitly maintaining it) by storing LCPs of consecutive points and performing longest-prefix matches.

Figure 5 shows a Patricia tree formed by endpoints s_1, t_1, s_2, t_2, ..., s_n, t_n, for a nondisjoint point intersection problem. Each edge of the Patricia tree corresponds to an ELCP. It is a 0-ELCP if it is a left edge and a 1-ELCP if it is a right edge. For every input range [s, t], we draw a *band* from s to LCP(s, t) and back to t. This band passes through some edges of the tree. It could pass an edge either from the left or the right sides. We say that the band's portions from s to LCP(s, t) pass through their edges on the right side. And the remaining portion of the band passes through its edges on the left side. For each edge, we count the number of bands passing through its left side (left count) and its right side (right count).

We also maintain another count, *Tcount*, for each edge (u, v) where u is the parent of v. If v is the right child of u, for *Tcount* we count the number of bands that pass through node u's right side but not through (u, v). Similarly, if v is the left child of u, for *Tcount* we count the number of bands that pass through u's left side but not through (u, v).

Here, we make the following observation: If query x lies in input range [s, t], then the path from x to the root must touch band(s, t) somewhere on the path. Considering the first node where the path intersects the band, we have two cases.

Case 1

The first point of intersection is not an internal node of the Patricia tree. This is the point where the path from x to root first touches the Patricia tree. If x hits band (s, t) on edge e from the left side, then we must have counted band (s, t) in the edge's left count. We have a similar result, if x hits edge e from the right side.

Case 2

The first point of intersection is an internal node of the Patricia tree. If the path from x to root hits band(s, t) along edge e, then we must have counted band(s, t) in the *Tcount* for edge e. This gives us the following algorithm for finding whether the path from x to the root hits a band.

Find the first edge, e, where the path from x to the root touches the tree. If the count for x's side (left or right) of e is nonzero, then it has touched a band. Otherwise, we simply check if any edge from e to the root has a nonzero *Tcount*. If so, there is an intersection; otherwise, there is none.

In terms of ELCPs, this method translates into the following. Find the longest ELCP that matches x. Check whether the count on x's side of ELCP is nonzero. If not, then check whether any ELCP that is a prefix of x has a nonzero *Tcount*. You can do this in one lookup by maintaining all the ELCPs with nonzero *Tcounts* in separate class C_2. You must maintain these counts during insertions and deletions.

To locate the prefixes during deletions, we further subdivide this class into two classes: one for prefixes ending in 0 and the other for those ending in 1. Alternately, we can point to these prefixes from their original copies in C_0 or C_1.

We provide a mechanism to compute prefix P's parent using a TCAM by performing two lookups. Let Q be the prefix obtained by dropping P's last bit. Look up Q01 ... 1 and Q10 ... 0 in class C_0 and C_1. Take the longer of the two lookup results.

It might seem that for insertions, while looking up all the ELCPs from s to LCP(s, t), you must perform 2log W lookups. But we can easily reduce this to log W by storing the prefix lengths looked up in the two TCAM classes.

For random points, the Patricia tree is well balanced[5] and so the number of LCPs on the path would be O(log n). So you can perform insertions in O(log n) time for random inputs.

Furthermore, for applications involving short, mostly scattered ranges with few overlaps, the LCPs would rarely be subprefixes of each other. In such cases, the insertions and deletions would take O(1) time because you only need to search up to the height of LCP(s, t).

As for the range intersection problem, we again use the reduction to the point intersection algorithm to obtain another algorithm with the same complexity.

These algorithms depend on longest-prefix matches in the TCAM. As before, we can eliminate this requirement by doubling the width of TCAM entries and performing a binary search for a longest-prefix lookup. Now the search time would be 2log(log W). For applications that can tolerate O(log W) time for searches and insertions, you can implement these algorithms in a CAM by searching all the prefixes for the longest-prefix match.

Figure 6 shows algorithm PI_OPT, which we derived based on this approach.

We provide a set of TCAM-based algorithms with varying time, space, and implementation complexities for sorting and searching. In particular, for the point and range intersection problems for disjoint ranges, we provide an algorithm that uses constant time and space for all operations, assum-

```
// ELCP→q : For 0-ELCPs store largest endpoint matching it
//          : For 1-ELCPs store smallest endpoint matching it
// ELCP→count[side], side could be left or right
//                          stores left count and right count.
// ELCP→Tcount


PI_point_lookup (p) {
    ELCP0 = Lookup p in class C0
    if (ELCP0 exists) and (p ≥ ELCP0→q)
        then return ELCP0
    ELCP1 = Lookup p in class C1
    if (ELCP1 exists) and (p ≤  ELCP1→q)
        then return ELCP1
}

Search (x) {
    ELCP = PI_point_lookup(x)
    if (x ≤ ELCP→q) and (ELCP→count[left] > 0) then
        return TRUE
    elseif (x ≥ ELCP→q) and (ELCP→count[right] > 0)
        return TRUE
    else {
        Lookup x in class C3
        if (hit) return TRUE
        else return FALSE
    }
}


// side is 0 if it is left side, else 1
subroutine_insert (r, len, side) {
    v = Point_lookup(r)
    P = longest_common_prefix(r, v)
    Insert P0, P1 in the TCAM class C0 and C1 resp
    if newly inserted then initialize all their counts to 0
    P0→ q = min(v,r)
    P1→ q = max(v,r)
    P = r
    while (P = longest LCP matching r of length less than P)
            and (length(P)-1 ≥  len ) {
        P→ count[side]++;
        Bit b = 0 if (P→q < r), 1 otherwise
        if (b == last bit of P)
            P→ q = r
        if (length(P)-1 > len) and (last bit of P! = side) {
            P′ = P with last bit inverted
            Lookup P′
            If it exists then
                increment its Tcount
            if (Tcount == 1)
                insert P′ in C2
        }
    }
}

insert (s, t) {
    P = longest_common_prefix (s,t);
    subroutine_insert (s, length(P), right);
    subroutine_insert (t, length(P), left);
}

subroutine_delete (r, len, side) {
    P = longest ELCP matching r obtained by doing 2 lookups
            in C0 and C1
    P→ count[side]--
    if(P→ count[left]==0 && P→count[right]==0) {
        d = (sibling_ of_P) →q
        delete P from TCAM;
        delete sibling of P from TCAM;
    }

    P = r;
    while (P = longest ELCP matching r of length less than P)
            and (length(P)-1 ≥ len ) {
        P→ count[side]--;
        if (P→ q == r)
            P→ q = d;
        if (length(P)-1 > len) and (last bit of P! = side) {
            P′ = Lookup sibling of P
            decrement P′→Tcount
            if (P′→Tcount == 0) then
                delete P′ from class C2
        }
    }
}

delete (s, t) {
    LCP = longest_common_prefix(s, t);
    subroutine_delete (s, length(LCP), right);
    subroutine_delete (t, length(LCP), left);
}
```

Figure 6. Algorithm PI_OPT.

ing an engine for longest-prefix matching. For the general case, we provide algorithms that require $O(1)$ time for search and $O(\log W)$ time for insert; they use $O(1)$ space. This enables maintaining a sorted list or a priority queue with $O(1)$ time for insertions and deletions. It would be interesting to find an algorithm for the general case that performs all operations in constant time and space. Also, for nondisjoint ranges, our algorithm does not report all the intersecting ranges but simply responds in a Boolean fashion, indicating whether or not there is an intersection. An interesting open problem would be to enhance this algorithm to report all the intersecting ranges. MICRO

References

1. D. Eppstein and S. Muthukrishnan, "Internet Packet Filter Management and Rectangle Geometry," *12th ACM-SIAM Symp. Discrete Algorithms*, ACM Press, 2001, pp. 827-835.
2. A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," *Proc. IEEE Infocom*, IEEE Press, 2000, pp. 1193-1202.
3. P. Gupta and N. McKeown, "Algorithms for

Packet Classification," *IEEE Network*, vol. 15, no. 2, Mar.-Apr. 2001, pp 24-32.

4. V. Srinivasan et al., "Fast and Scalable Layer Four Switching," *ACM Computer Comm. Rev.*, vol. 28, no. 4, Sept. 1998, pp. 191-202.
5. D.E. Knuth, *The Art of Computer Programming: Sorting and Searching*, 2nd ed., Addison-Wesley, 1998.
6. R. Sedgewick, *Algorithms in C*, Addison-Wesley, 1990, p. 253.
7. D.E. Willard, "Log-Logarithmic Worst-Case Range Queries are Possible in Space," *Information Processing Letters*, vol. 17, no. 2, 1983, pp. 81-84.
8. A. Andersson et al., "Sorting in Linear Time?" *Proc. 27th Ann. ACM Symp. Theory of Computing*, ACM Press, 1995, pp. 427-436.
9. D. Shah and P. Gupta, "Fast Updates on Ternary-CAMs for Packet Lookups and Classification," *IEEE Micro*, vol. 21, no. 1, Jan.- Feb. 2001, pp. 36-47.

Rina Panigrahy is a software engineer at Cisco Systems. His research interests include theoretical algorithms, TCAM algorithms for forwarding, ACLs, power reduction, RAM-based algorithm for IPv6, and regular expression processing for L7 switching. Panigrahy has a master's degree from MIT, where he worked on Web caching technology that became the basis for founding Akamai Technologies.

Samar Sharma is a senior software engineer at Cisco Systems. His work experience includes several products involving IP forwarding, TCAMs, network processors, access lists, and storage networking. Sharma has a BTech in computer science from the Indian Institute of Technology, Delhi, and an MS from the University of Maryland. He has coauthored several patent applications and research papers while at Cisco.

Direct questions and comments about this article to Samar Sharma, 655 South Fair Oaks Ave., #E204, Sunnyvale, CA 94086; ssharma @cisco.com.

For further information on this or any other computing topic, visit our Digital Library at http://computer.org/publications/dlib.