INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS
Int. J. Commun. Syst. 2008; **21**:115–133
Published online 8 March 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/dac.879

On a trie partitioning algorithm for power-efficient TCAMs

Haibin Lu*, †

Department of Computer Science, University of Missouri, Columbia, MO 65211, U.S.A.

SUMMARY

Internet routers conduct routing table (RT) lookup based on the destination IP address of the incoming packet to decide which output port to forward the packet. Ternary content-addressible memories (TCAM) uses parallelism to achieve lookup in a single cycle. One of the major drawbacks of TCAM is its high-power consumption. Trie-based architecture has been proposed to reduce TCAM power consumption. The idea is to use an index TCAM to select one of many data TCAM blocks for lookup. However, power reduction is limited by the size of the index TCAM, which is always enabled for search. In this paper we develop a simple but effective trie-partitioning algorithm to reduce the index TCAM size, which achieves better reduction in power consumption, and at the same time guarantees full TCAM space utilization. We compared our algorithm (LogSplit) with PostOrderSplit (*IEEE INFOCOM*, 2003). For two real-world RTs (AADS and PAIX), the size of the index TCAM generated by LogSplit is 55–70% of that generated by PostOrderSplit; the largest power reduction factor of LogSplit is 41 for AADS and 68 for PAIX, while the largest power reduction factor of PostOrderSplit is 33 for AADS and 52 for PAIX. The improvement is even more significant in the worst case: the size of the index TCAM generated by LogSplit is 18–30% of that generated by PostOrderSplit for IPv4, and less than 1% of that generated by PostOrderSplit for IPv6; the largest power reduction factor of LogSplit is 173 for both IPv4 and IPv6, while the largest power reduction factor of PostOrderSplit is only 82 for IPv4 and 41 for IPv6. Copyright © 2007 John Wiley & Sons, Ltd.

Received 12 May 2006; Revised 8 January 2007; Accepted 8 January 2007

KEY WORDS: TCAM; router; IP lookup; packet forwarding; power-efficient

1. INTRODUCTION

Packet forwarding uses a routing table (RT) to find the next hop for each incoming packet. Each entry in the RT is a pair of the form (prefix, next hop). The packet-header field used is the

*Correspondence to: Haibin Lu, Department of Computer Science, University of Missouri, Columbia, MO 65211, U.S.A.
†E-mail: luhaibin@missouri.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: CNS-0423386

destination IP address of a packet. When there are several prefixes that match the destination IP address of a packet, the longest-prefix match rule is used to select one of them. Let $lmp(p)$ be the longest matching prefix in the RT for prefix p (note that IP address is a prefix with length W, where W is equal to 32 for IPv4 and 128 for IPv6). When $RT = \{*, 0*, 1*, 001*, 0000*, 0010*, 00001*\}$ (we omit the action part since it does not affect the algorithms we will discuss), $0000*$ is matched by $0*$ and $0000*$; $lmp(0000*) = 0000*$, since the length of $0000*$ is four and the length of $0*$ is one; similarly, $lmp(000*) = 0*$. Assume $W = 6$, the destination IP address 000011 is matched by $0*$, $0000*$ and $00001*$, so $lmp(000011) = 00001*$.

Ternary content-addressible memories (TCAMs) use parallelism to perform lookup in a single cycle [1]. Each memory cell of a TCAM may be set to one of three states: 0, 1, and 'don't care'. The prefixes are stored in a TCAM in descending order of prefix length. Assume that each word of the TCAM has 32 cells. The prefix 10* is stored in a TCAM word as 10???...?, where ? denotes a 'don't care' and there are 30 ?s in the given sequence ($W = 32$ for IPv4). To do a longest-prefix match, the destination IP address is matched, in parallel, against every TCAM entry and the first (i.e. longest) matching entry is reported by the TCAM arbitration logic. So, using a TCAM and a sorted-by-length linear list, the longest matching-prefix can be determined in $O(1)$ time. A prefix may be inserted or deleted in $O(W)$ time [2]. Although TCAMs provide a simple and efficient solution for RT lookup, this solution has very high-power consumption (up to 15 W for a current high-density TCAM). High-power consumption increases cooling cost and limits the number of ports in a single board.

Zane *et al.* [3] use two architectures, bit-selection based and trie based, to reduce TCAM power consumption. The idea is to reduce the number of TCAM entries that are checked during lookup. The schemes use multiple TCAMs or a single TCAM with multiple blocks. The bit-selection-based architecture assumes that the prefix length is between 16 and 24 (thus not suitable for IPv6) and is less efficient in reducing power consumption than the trie-based architecture. In the trie-based architecture, an index TCAM is searched first and the result is used to select one of many data TCAM blocks. The RT is partitioned into many groups. The partitioning algorithms include SubtreeSplit and PostOrderSplit [3]. SubtreeSplit may waste up to half of data TCAM entries, and thus is too space-inefficient to be practical. PostOrderSplit fully utilizes each data TCAM block except the last one. However, it generates a large index TCAM. Each data TCAM block contributes up to $W + 1$ entries to the index TCAM. Thus, for search engine with 256 data TCAM blocks,

Table I. Notation.

Symbol	Definition
RT	Routing table
n	The number of prefixes in RT
m	The number of entries in a data TCAM block
d	Destination IP address
$lmp(p)$	The longest matching prefix in RT for prefix p
$x.left$	Left child of node x
$x.right$	Right child of node x
$x.parent$	Parent of node x
$subtrie(x)$	Subtree rooted at node x (*including x itself*)
$prefix(x)$	The prefix denoting the path from the root to node x
$count(x)$	The number of prefixes $\in RT$ stored in $subtrie(x)$
$cp(x)$	Covering prefix of node x. $cp(x) = lmp(prefix(x))$

DOI: 10.1002/dac

the index TCAM has $256 \times 33 = 8446$ entries (270 kbits) for IPv4 and $256 \times 129 = 33\,024$ entries (4.3 Mbits) for IPv6. Since the index TCAM is always on, its power consumption is significant, especially for IPv6 router tables (a current 16 Mbits TCAM has a power consumption of up to 15 W). Since the router hardware designers have to use the worst-case bound to determine the size of an index TCAM and the power budget, it is necessary to reduce the worst-case size of the index TCAM.

The contribution of this paper is a new trie-partitioning algorithm, LogSplit, in which each data TCAM block adds at most $\log_2 m$ entries to the index TCAM, where m is the number of entries in one data TCAM block.

The rest of the paper is organized as follows. Section 2 gives the background of the trie-based architecture and discusses the PostOrderSplit algorithm [3]. Section 3 proposes and analyses the LogSplit algorithm. Section 4 compares SubtreeSplit [3], PostOrderSplit [3] and our LogSplit. Section 5 summarizes related work, and Section 6 gives the conclusion. Table I lists the notations we will use in this paper.

2. BACKGROUND

The trie-based architecture [3] is shown in Figure 1. The data TCAM consist of multiple TCAM blocks (for example, 5-bit ID is used to select one of 32 TCAM blocks). Index SRAM stores the block ID. Index TCAM is searched first using the destination IP address, and the index of the longest matching entry is used to retrieve the block ID from the index SRAM. Then the corresponding data TCAM block is enabled for search. Finally, the index of the longest matching entry in the selected data TCAM block is used to retrieve the next hop information stored at the data SRAM. Since only the index TCAM and one of the data TCAM blocks are enabled for search, power consumption is greatly reduced.

To populate data TCAM blocks and index TCAM, a one-bit trie is first constructed. Suppose $RT = \{*, 0*, 1*, 001*, 0000*, 0010*, 00001*\}$. The corresponding one-bit trie is shown in Figure 2. At an internal node of a one-bit trie, the branch is determined by the next leading bit of the input prefix. Bit 0 leads to the left child and bit 1 to the right child. The path from the root to node x can be represented by $prefix(x)$. $prefix(root) = *$, $prefix(root.left) = 0*$, $prefix(root.right) = 1*$, and so on. In general, $prefix(x)$ is 0 concatenated with $prefix\ (x.parent)$ if x is the left child of $x.parent$, and 1 concatenated with $prefix(x.parent)$ if x is the right child of

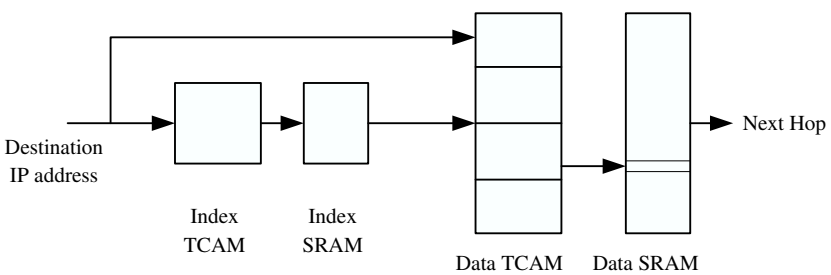

Figure 1. Trie-based architecture.

DOI: 10.1002/dac

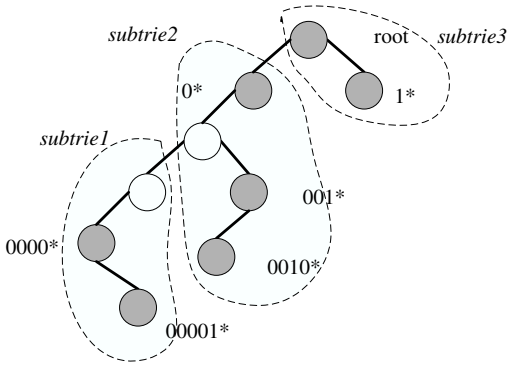

Figure 2. One-bit trie for prefix set $RT = \{*, 0*, 1*, 001*, 0000*, 0010*, 00001*\}$. The shaded nodes store prefixes $\in RT$. Note that all leaf nodes are shaded.

$x.parent$. Since $prefix(x)$ is unique, we can use $prefix(x)$ to refer to node x. For instance, '$node\ *$' refers to the root and '$node1*$' refers to the right child of the root. The shaded nodes in Figure 2 store prefixes from the RT. Using a one-bit trie, the lmp of any prefix p is easy to figure out. The $lmp(p)$ is p if $node\ p$ is shaded, or it is the $prefix$ value of the nearest shaded ancestor of $node\ p$. For example, $lmp(00001*)$ is $00001*$ and $lmp(000*)$ is $0*$.

Second, one-bit trie is partitioned. At each step, a qualified subtrie is pruned completely. The qualification of a subtrie is determined by partitioning algorithms, e.g. SubtreeSplit [3], PostOrder-Split [3], and our LogSplit. In Figure 2, $subtrie1$ is pruned first, then $subtrie2$, and finally the remaining trie. Note that the subtries pruned are disjoint and their union is the original trie.

Third, for each subtrie pruned (let rt be the root of this subtrie), we associate a covering prefix $cp(rt)$ with rt. The covering prefix of node x, $cp(x)$, is defined as $lmp(prefix(x))$. For example, in Figure 2, the covering prefix of $node\ 000*$ is $0*$, the covering prefix of $node\ 0*$ is $0*$, and the covering prefix of $node\ *$ is $*$. Note that if $prefix(rt) \in RT$, then $cp(rt)$ is equal to $prefix(rt)$. Since RT contains default prefix *, $cp(rt)$ always exists. The purpose of associating $cp(rt)$ with every pruned subtrie is to ensure that a subtrie, when searched, can always return the $lmp(d)$. Suppose the destination address $d = 000100$ (assume $W = 6$). The $subtrie1$ in Figure 2 is searched for $lmp(d)$. Note that $subtrie1$ was pruned already and the original trie is not available for search. Searching $subtrie1$ would return $null$ if $cp(node\ 000*)$ was not associated with $subtrie1$. With the covering prefix, $lmp(000100) = cp(node\ 000*) = 0*$. It is easy to see that in order to return the correct $lmp(d)$, a proper subtrie needs to be located first. For example, searching $lmp(000100)$ in $subtrie3$ will return a wrong result. The index TCAM discussed in the next paragraph is used to locate a proper subtrie.

Fourth, each pruned subtrie contributes one prefix, the $prefix$ of the root of this subtrie, to the index TCAM. In Figure 2, $subtrie1$ contributes 000*, $subtrie2$ contributes 0*, and $subtrie3$ contributes *. Thus, the index TCAM contains prefixes $\{000*, 0*, *\}$. The longest matching prefix in the index TCAM leads to a proper subtrie for search. For example, when the destination IP address $d = 001000$ (assume $W = 6$), $0*$ is the longest matching prefix in the index TCAM, $subtrie2$ is located and searched, and $0010*$ is returned as the longest matching prefix. When $d = 000100$ (assume $W = 6$), $000*$ is the longest matching prefix in the index TCAM, $subtrie1$ is located and searched, and $0*$ is returned as the longest matching prefix.

DOI: 10.1002/dac

Finally, we use data TCAM blocks to store these pruned subtries and their covering prefixes. SubtreeSplit [3] uses a separate TCAM block for each subtrie. So, under SubtreeSplit, the first data TCAM block contains {0000∗, 00001∗, 0∗} (note that 0∗ is the covering prefix), the second data TCAM block contains {0010∗, 001∗, 0∗}, and the third data TCAM block contains {∗, 1∗}. PostOrderSplit [3] and our LogSplit may store multiple subtries in one data TCAM block.

2.1. SubtreeSplit and PostOrderSplit

Zane *et al.* [3] propose SubtreeSplit and PostOrderSplit for partitioning trie. Let us first define $count(x)$ for trie node x. The $count(x)$ is the number of prefixes $\in RT$ stored in $subtrie(x)$.

$$count(x) = \begin{cases} count(x.left) + count(x.right) & \text{if } x \neq null \text{ and } prefix(x) \notin RT \\ count(x.left) + count(x.right) + 1 & \text{if } x \neq null \text{ and } prefix(x) \in RT \\ 0 & \text{if } x = null \end{cases} \quad (1)$$

Figure 3 gives the *count* values of all trie nodes in Figure 2. Note that the *count* value of a leaf node is always one.

SubtreeSplit traverses trie in post order and checks every node encountered. When $count(x) \geqslant \lceil m/2 \rceil$ and $count(x.parent) > m$, where x is the current node, $subtrie(x)$ is pruned and stored in a data TCAM block. The algorithm then moves on to a new data TCAM block. The advantage of SubtreeSplit is that one data TCAM block contributes one prefix to the index TCAM. However, in the worst case, a data TCAM block is only 50% full.

To fill data TCAM blocks, the PostOrderSplit algorithm is proposed. PostOrderSplit allows multiple subtries in each data TCAM block. It traverses the one-bit trie in post order and prunes $subtrie(x)$ when either $count(x) = s$ or $(count(x) < s \text{ and } count(x.parent) > s)$, where x is the current node and s is the number of empty entries left in the current data TCAM block. The $subtrie(x)$ is added to the current data TCAM block and the s value is updated. When the current data TCAM block is full, the algorithm moves on to a new data TCAM block.

It is easy to see that PostOrderSplit may prune subtries with only a few prefixes each. This happens when $count(x)$ is small and $count(x.parent) > s$, i.e. the *count* value of the sibling of node x is big. Figure 4 gives an example of the execution of the PostOrderSplit algorithm. The algorithm prunes four subtries with one prefix each. In the worst case, PostOrderSplit needs $W + 1$

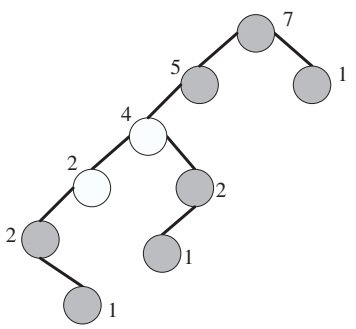

Figure 3. *count* values of the trie nodes in Figure 2.

DOI: 10.1002/dac

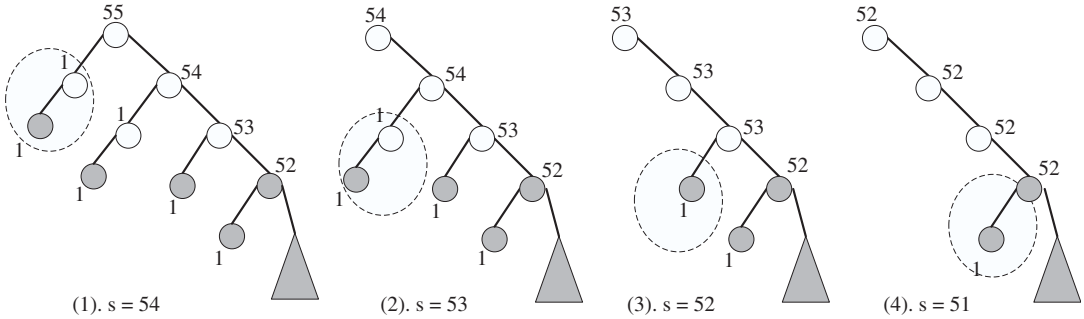

Figure 4. PostOrderSplit may prune a subtrie containing a few prefixes each. The number beside node x is $count(x)$. The shaded node x has $prefix(x) \in RT$.

subtries to fill one data TCAM block [3]. Since each subtrie contributes one prefix to the index TCAM, one data TCAM block may contribute up to $W + 1$ prefixes to the index TCAM.

The drawback of PostOrderSplit is that the size of the index TCAM may be big. For example, when 256 data TCAM blocks are used, the index TCAM needs 8448 entries (270 kbits) for IPv4 and 33 024 entries (4 Mbits) for IPv6, in the worst case. Large index TCAM size limits the ability to reduce power consumption, since the index TCAM is always on for search. The power consumption of a 270 kbit TCAM may be affordable, but the power consumption of a 4 Mbit TCAM is definitely significant, since a current 16 Mbit TCAM consumes up to 15 W when all entries are enabled. One may argue that the case in Figure 4 is rare. However, hardware designers have to decide on a size for the index TCAM. The question then becomes, what is the right size for index TCAM? Since RTs are frequently updated and are growing over time, the worst case is usually used to allocate the power budget and the index TCAM.

The average size of an index TCAM is smaller than that in the worst case. We can certainly divide the index TCAM into blocks and enable only those blocks that contain prefixes. However, we may need to turn on several index TCAM blocks, and extra hardware is needed to select one output from the many outputs of these index TCAM blocks. Beside, this approach does not reduce the worst-case size of the index TCAM.

3. LOGSPLIT: A NEW TRIE-PARTITIONING ALGORITHM

In this section we present a new algorithm to partition a one-bit trie. This algorithm greatly reduces the size of the index TCAM. More specifically, each data TCAM block contributes at most $\log_2 m$ entries to the index TCAM, where m is the number of entries in a data TCAM block. The algorithm works by finding a subtrie with at least $\lceil e/2 \rceil$ prefixes at each iteration, where e is the number of empty entries in the current data TCAM block. Initially $e = m$ and is updated at each iteration.

3.1. The algorithm

We first show a property of the *count* value of a trie node as computed using Equation (1). Lemma 3 is the key for our trie-partitioning algorithm.

DOI: 10.1002/dac

Lemma 1
If $x \neq null$, then one of the following is true.

1. $count(x) = count(x.left)$.
2. $count(x) = count(x.right)$.
3. $count(x) > count(x.left)$ and $count(x) > count(x.right)$.

Proof
This lemma follows from Equation (1). □

Lemma 2
If $x \neq null$, then there is a node z in $subtrie(x)$ such that ($count(z) = count(x)$ and $count(z) > count(z.left)$ and $count(z) > count(z.right)$). Note that z may be x itself.

Proof
Let $z = x$. The algorithm walks down the trie. If $count(z) = count(z.left)$, then the algorithm moves to $z.left$. If $count(z) = count(z.right)$, then the algorithm moves to $z.right$. Otherwise, the algorithm returns z. Since $count(u) = 1 > count(u.left) = count(u.right) = 0$ for any leaf node u, the algorithm will always terminate and return z. According to Lemma 1, $count(z) > count(z.left)$ and $count(z) > count(z.right)$. □

Lemma 3
If $count(x) > s$ where s is a positive integer, then there is a node y in $subtrie(x)$ such that $\lceil s/2 \rceil \leqslant count(y) \leqslant s$.

Proof
Suppose there is no node y in $subtrie(x)$ such that $\lceil s/2 \rceil \leqslant count(y) \leqslant s$. Then, for any node y in $subtire(x)$, $count(y)$ is either greater than s or smaller than $\lceil s/2 \rceil$. Let cz be the smallest $count$ value larger than s. Let u be the node such that $count(u) = cz$ (if there are more than one such node, we arbitrarily pick one). According to Lemma 2, there is a node z in $subtrie(u)$ such that

$$count(z) = cz \quad \text{and} \quad count(z) > count(z.left) \quad \text{and} \quad count(z) > count(z.right)$$

Under our assumption, there is no node y such that $\lceil s/2 \rceil \leqslant count(y) \leqslant s$. Hence, $count(z.right) < \lceil s/2 \rceil$ and $count(z.left) < \lceil s/2 \rceil$ must be true, since $count(z) (= cz)$ is the smallest $count$ value larger than s. However, from Equation (1), $count(z) \leqslant count(z.left) + count(z.right) + 1 \leqslant \lceil s/2 \rceil - 1 + \lceil s/2 \rceil - 1 + 1 = 2\lceil s/2 \rceil - 1 \leqslant s$. This contradicts $count(z) > s$. □

Figure 5 gives the algorithm to find node y as described in Lemma 3. The parameter s is a positive integer. The algorithm assumes $count(root) \geqslant \lceil s/2 \rceil$. Since the algorithm starts at the root of the trie and walks down the trie in binary-search fashion, it takes $O(W)$ time to find node y. Let $s = 3$ and use Figure 3 as an example. The algorithm starts at the root, checks $node\ 0*$, $node\ 00*$, and then $node\ 000*$. Since $s \geqslant count(node\ 000*) = 2 \geqslant \lceil s/2 \rceil$, $node\ 000*$ is returned.

Theorem 1
Given that $count(root) \geqslant \lceil s/2 \rceil$ where s is a positive integer, the algorithm in Figure 5 correctly returns node y such that $\lceil s/2 \rceil \leqslant count(y) \leqslant s$.

DOI: 10.1002/dac

```
//precondition: count(root) ≥ ⌈s/2⌉.
Algorithm getNodeY(s){
  //return node y such that ⌈s/2⌉ ≤ count(y) ≤ s.
  x = root;
  while(count(x) > s){
    if(count(x.left)≥ ⌈s/2⌉)
        x = x.left;
    else
        x = x.right;
  }
  return x;
}
```

Figure 5. Algorithm to find node y in the one-bit trie such that $\lceil s/2 \rceil \leqslant count(y) \leqslant s$.

Proof

The algorithm starts at the root. If $count(root) \leqslant s$, the *while* loop is not executed and the algorithm returns $root$. Since $\lceil s/2 \rceil \leqslant count(root)$, we find y.

If $count(root) > s$, the while loop is executed. The algorithm walks down the trie. If $count(x.left) \geqslant \lceil s/2 \rceil$, then either $count(x.left) > s$ or $\lceil s/2 \rceil \leqslant count(x.left) \leqslant s$. If $count(x.left) > s$, there must be a node y in $subtrie(x.left)$ according to Lemma 3. The algorithm moves to $x.left$. If $\lceil s/2 \rceil \leqslant count(x.left) \leqslant s$, the while loop terminates and we find y.

If $\lceil s/2 \rceil > count(x.left)$, we goes to $x.right$. The $count(x.right)$ is not smaller than $\lceil \frac{s}{2} \rceil$; otherwise, $count(x) \leqslant count(x.left) + count(x.right) + 1 \leqslant \lceil s/2 \rceil - 1 + \lceil s/2 \rceil - 1 + 1 = 2\lceil s/2 \rceil - 1 \leqslant s$. Therefore, either $count(x.right) > s$ or $\lceil s/2 \rceil \leqslant count(x.right) \leqslant s$. If $count(x.right) > s$, there must be a node y in $subtrie(x.right)$ according to Lemma 3. The algorithm moves to $x.right$. If $\lceil s/2 \rceil \leqslant count(x.right) \leqslant s$, the while loop terminates and we find y.

Since such y must exist according to Lemma 3, the while loop will eventually terminate. □

LogSplit algorithm (Figure 6) uses the $getNodeY$ function. The algorithm continuously finds and prunes subtrees until the remaining trie contains no more than m prefixes (Line 02). The algorithm uses $getNodeY$ (Line 07) to find a $subtrie(y)$ with at least $\lceil e/2 \rceil$ prefixes, where e is the number of empty entries in the the current data TCAM block. The algorithm prunes $subtrie(y)$ and adds prefixes in $subtrie(y)$ to the current bucket. If $prefix(y) \notin RT$, $cp(y)$ is added to the current bucket. If $prefix(y) \in RT$, there is no need to add $cp(y)$, since $cp(y)$ is equal to $prefix(y)$ in this case. The value e is updated accordingly and is reduced by at least half of its previous value. The inner loop continues until the current bucket is full (Line 06). Note that the algorithm reduces e by one (Line 05) after setting e to m (Line 04) in order to have an empty entry for $cp(y)$ when $prefix(y) \notin RT$. This may leave one entry unfilled in each data TCAM block.

Figure 7 gives the execution of LogSplit on the trie in Figure 4. The algorithm prunes a subtrie with 53 prefixes. The covering prefix is also added to the bucket. The algorithm then terminates. We will use Figure 2 as another example. Let $m = 4$. The algorithm prunes *subtrie*1 first, then *subtrie*2. Finally, the remaining trie, *subtrie*3, is pruned (Lines 17–19 in Figure 6).

DOI: 10.1002/dac

```
Algorithm LogSplit(m) {
01   i = 0;
02   while(count(root) > m){ // root is the root of the original trie.
03       i = i + 1; // allocate a new bucket. bucket_i will be assigned to the i-th data TCAM
         block.
04       e = m; // e is the number of empty entries.
05       e = e − 1; // allocate an entry for a potential covering prefix.
06       while(e > 0){
07          y = getNodeY(e);
08          Prune subtrie(y); Add prefixes in subtrie(y) to bucket_i; e = e − count(y);
09          if(prefix(y) ∉ RT)
10             {Add cp(y) to bucket_i; e = e − 1; }
11          Add prefix(y) to the index TCAM;
12          for (each node x along the path from root to y.parent)
13             count(x) = count(x) − count(y);
14       }
15   }
16   if( count(root) > 0){
17      i = i + 1; // allocate a new bucket.
18      Prune subtrie(root); Add prefixes in subtrie(root) to bucket_i;
19      Add prefix(root) to index TCAM;
20   }
21   return bucket_1, bucket_2, ···, bucket_i;
}
```

Figure 6. The new algorithm to partition a one-bit trie.

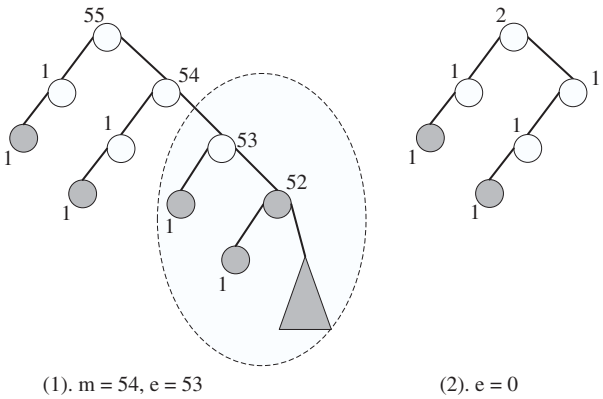

Figure 7. The execution of LogSplit on the trie in Figure 4. $m = 54$.

Theorem 2

Let m be the number of entries in a data TCAM block. For each data TCAM block, the $LogSplit$ algorithm in Figure 6 adds at most $\log_2 m$ prefixes into the index TCAM and generates at most $\log_2 m$ covering prefixes.

Proof

The theorem is true since the value e is reduced by at least half of its previous value at each iteration (Line 08 in Figure 6). □

DOI: 10.1002/dac

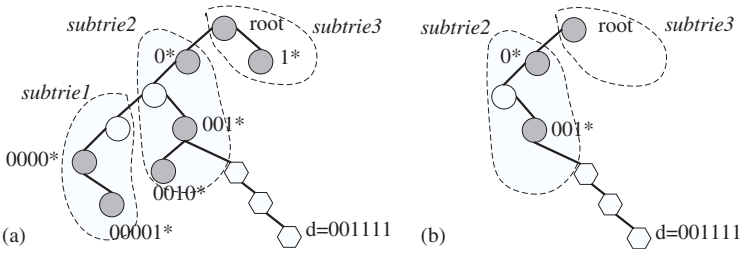

Figure 8. The destination address $d = 001111$. Shaded nodes store the prefixes in RT.

Theorem 3

Searching the index TCAM and the data TCAM blocks populated by the LogSplit algorithm returns $lmp(d)$ for any destination address d.

Proof

LogSplit partitions the original trie. Let$\{ST_1, ST_2, \ldots, ST_l\}$ be the resulting subtries, where ST_i is a set of trie nodes. This set has the following properties:

$P1$: These subtries are disjoint. That is, $ST_i \cap ST_j = \emptyset$ for any $i \neq j$.

$P2$: The union of these subsets is equal to the set of the nodes in the original trie.

$P3$: If a node $x \in ST_i$ is an ancestor of a node $y \in ST_j$ $(i \neq j)$, then u is not a descendant of v for any node $u \in ST_i$ and any node $v \in ST_j$. This is because a complete subtrie is removed whenever LogSplit prunes a subtrie (Lines 08 and 18 in Figure 6).

With these three properties, we first give an intuitive proof of the theorem. We use the one-bit trie in Figure 2 as an example. Let $d = 001111$. We complete the path that leads to $node\ d$ (Figure 8(a)). We use hexagons to depict the nodes we just added. The nodes in the original trie are depicted by circles. We then remove those nodes in Figure 8(a) that are not ancestors of $node\ d$. The remaining trie is shown in Figure 8(b). The remaining nodes form a chain from the root to $node\ d$. The chain contains all prefixes in RT that match d. The nodes that store these prefixes are shaded. The chain (excluding hexagonal nodes) was partitioned into disjoint segments by LogSplit. The $prefix$ value of the root of each chain segment was added to the index TCAM. Searching in the index TCAM leads to the chain segment that is the lowest in the original trie. Searching this chain segment (*subtrie*2 in this case) returns the correct $lmp(d) = 001*$, since there are shaded nodes in this chain segment. It is possible that the chain segment contains no shaded nodes. Let $d = 000110$. After we repeat the process just described, we get a chain as shown in Figure 9(b). Searching in the index TCAM leads to a chain segment (*subtrie*1 in this case). There are no shaded nodes in this chain segment. However, the covering prefix of the root of this chain segment ($0*$ in this case) is saved together with *subtrie*1 and is the longest matching prefix for d.

Below is the formal proof. Given a destination IP address d, the longest matching prefix in the index TCAM leads to subtrie ST_i with rt_i as its root. Note that this longest matching prefix in the index TCAM is $prefix(rt_i)$. Since we associated $cp(rt_i)$ with rt_i if $prefix(rt_i) \notin RT$, searching ST_i always returns a matching prefix. Let p_1 be the longest matching prefix in $ST_i \cup \{cp(rt_i)\}$. We argue that no matching prefix exists in RT that is longer than p_1. Suppose there is a longer matching prefix than p_1. We will call it p_2. Then $node\ p_2$ is a descendant of $node\ p_1$. We consider two cases: $node\ p_1 \in ST_i$ and $node\ p_1 \notin ST_i$.

DOI: 10.1002/dac

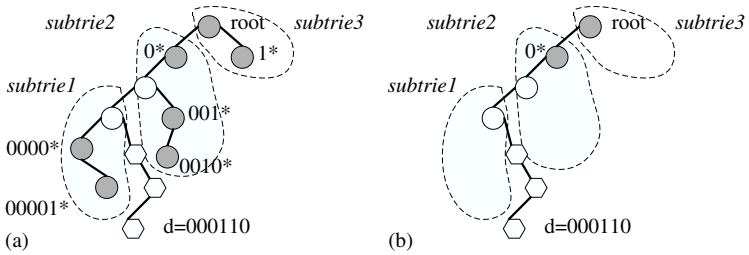

Figure 9. The destination address $d = 000110$. Shaded nodes store the prefixes in RT.

(1) $node\ p_1 \in ST_i$. That is, $node\ p_1$ is an actual node inside the subtrie ST_i.

  (1.1) $node\ p_2 \in ST_i$. Then searching $subtrie(rt_i)$ will return p_2 instead of p_1. This is a contradiction.

  (1.2) $node\ p_2 \notin ST_i$. That is, $node\ p_2$ is in a different subtrie, say, ST_j (let rt_j be the root of this subtrie). Note that $node\ p_2$ is a descendant of $node\ p_1$. From the third property of the set $\{ST_1, ST_2, \ldots, ST_l\}$, we know that rt_j is a descendant of rt_i. Hence, in the index TCAM, $prefix(rt_j)$ is a longer matching prefix for d than $prefix(rt_i)$. This is a contradiction.

(2) $node\ p_1 \notin ST_i$. That is, p_1 is $cp(rt_i)$ and the $node\ p_1$ is an ancestor of rt_i. Note that node $cp(rt_i)$ may not be an actual node inside ST_i. For instance, in Figure 2, the covering prefix of the root of *subtrie*1 is $0*$.

  (2.1) $node\ p_2 \in ST_i$. Similar to Case (1.1).

  (2.2) $node\ p_2 \notin ST_i$. That is, $node\ p_2$ is in a different subtrie, say, ST_j (let rt_j be the root of this subtrie). Since $node\ p_1$ is an ancestor of rt_i, rt_j may be a descendant of rt_i or an ancestor of rt_i.

    (2.2.1) rt_j is a descendant of rt_i. Since p_2 matches d and $p2 \subseteq prefix(rt_j)$, $prefix(rt_j)$ matches d. Thus, in the index TCAM, $prefix(rt_j)$ is a longer matching prefix for d than $prefix(rt_i)$. This is a contradiction.

    (2.2.2) rt_j is a ancestor of rt_i. The result of searching $lmp(d)$ in the index TCAM tells us that $prefix(rt_i)$ matches d. Note that p_2 also matches d. From the third property of the set $\{ST_1, ST_2, \ldots, ST_l\}$, we know that $node\ p_2$ is an ancestor of rt_i. Hence, the covering prefix of rt_i is p_2 instead of p_1. This is a contradiction. □

3.2. Time complexity

The algorithm makes at most $\log_2 m$ iterations to fill each bucket and each iteration invokes $getNodeY()$, which takes $O(W)$ time each. Hence, it takes $O(W \log_2 m)$ time to fill each bucket, excluding the time spent adding prefixes in the subtrie to the current bucket. The total time of adding the prefixes to the buckets is the size of the trie, which is $O(nW)$. Therefore, the overall time complexity of LogSplit is $O(nW + kW \log_2 m) = O(nW)$, where $k = \lfloor (n+m)/(m - \log_2 m) \rfloor$

DOI: 10.1002/dac

is the total number of data TCAM blocks (See Section 4.2). The time complexities of SubtreeSplit and PostOrderSplit [3] are also $O(nW)$.

4. EXPERIMENT

In Section 4.1, we present the experiment results of applying SubtreeSplit, PostOrderSplit and LogSplit to the RTs we obtained. These results assumes that RTs are static. However, in practice, RTs change over time and the distribution of the prefixes also changes. So the size of the index TCAM that is generated by these algorithms for a single RT cannot be reliably used as a guideline to decide the actual size of the index TCAM. Instead, the worst-case scenario is usually used to guide the design. We discuss the worst-case performance of these three algorithms in Section 4.2.

4.1. Performance on two routing tables

The IPv4 RTs we used are from [4]. The RT AADS was obtained on 22 November 2001, while PAIX was obtained on 13 September 2000. AADS has 31 828 prefixes and PAIX has 85 988 prefixes. We also tested other RTs such as Pb and MaeWest. The results are similar to that on AADS and PAIX. Due to space limitations, we only list the results on PAIX and AADS.

The time to build the trie and run the algorithms (Table II) is about 110 ms for AADS and about 250 ms for PAIX on a Pentium 4 1.5 GHz PC. The run time listed in Table II is the average of 10 run times. The time to run each algorithm changes little when m changes, because the time to build the trie and add prefixes to buckets dominates the time to find subtries. The codes of all three algorithms are not optimized for speed. We use recursion to simplify the codes. As shown in Table II, the run time is quite acceptable.

Given the size of a data TCAM block, m, we obtained the number of data TCAM blocks needed (Table III, plotted in Figure 10), the number of the index prefixes generated (Table IV, plotted in Figure 11), and power reduction achieved (Table V, plotted in Figure 12).

Table III lists the number of data TCAM blocks generated by these three algorithms *versus* the size of a data TCAM block. The number of data TCAM blocks used by SubtreeSplit is 25–38% (32% on average) more than that used by PostOrderSplit or LogSplit. PostOrderSplit and LogSplit fill all data TCAM blocks except the last block. They use the same number of data TCAM blocks except when $m = 128$. When $m = 128$, PostOrderSplit uses one more block than LogSplit. This is because PostOrderSplit generates more covering prefixes.

Table II. Run time (in milliseconds) *versus* the size of a data TCAM block.

	m	128	256	512	1024	2048	4096
AADS	SubtreeSplit	107	107	107	108	110	109
	PostOrderSplit	112	111	113	112	113	113
	LogSplit	112	111	113	112	113	113
PAIX	SubtreeSplit	261	266	263	265	263	276
	PostOrderSplit	250	249	251	251	252	253
	LogSplit	250	249	251	251	252	253

DOI: 10.1002/dac

Table III. The number of data TCAM blocks *versus* the size of a data TCAM block.

	m	128	256	512	1024	2048	4096
AADS	SubtreeSplit	331	169	83	40	21	11
	PostOrderSplit	252	125	63	32	16	8
	LogSplit	251	125	63	32	16	8
PAIX	SubtreeSplit	887	441	226	114	56	28
	PostOrderSplit	680	338	169	85	43	22
	LogSplit	679	338	169	85	43	22

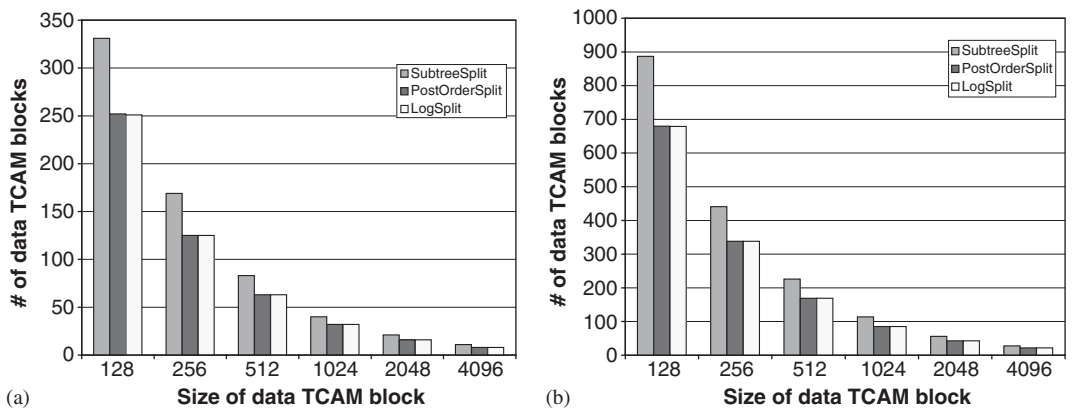

Figure 10. The number of data TCAM blocks *versus* the size of a data TCAM block: (a) AADS and (b) PAIX.

Table IV. The number of the index prefixes *versus* the size of a data TCAM block.

	m	128	256	512	1024	2048	4096
AADS	SubtreeSplit	331	169	83	40	21	11
	PostOrderSplit	1671	885	458	250	135	64
	LogSplit	920	530	302	153	83	45
PAIX	SubtreeSplit	887	441	226	114	56	28
	PostOrderSplit	4096	2175	1216	633	346	185
	LogSplit	2419	1322	752	410	219	122

Table IV lists the number of the index prefixes generated by these three algorithms *versus* the size of a data TCAM block. Though SubtreeSplit generates the least number of index prefixes (one per block), it uses many more data TCAM blocks (see Table III). As expected, LogSplit generates much fewer index prefixes than PostOrderSplit. The number of index prefixes generated by LogSplit is from 55 to 70% of that generated by PostOrderSplit.

DOI: 10.1002/dac

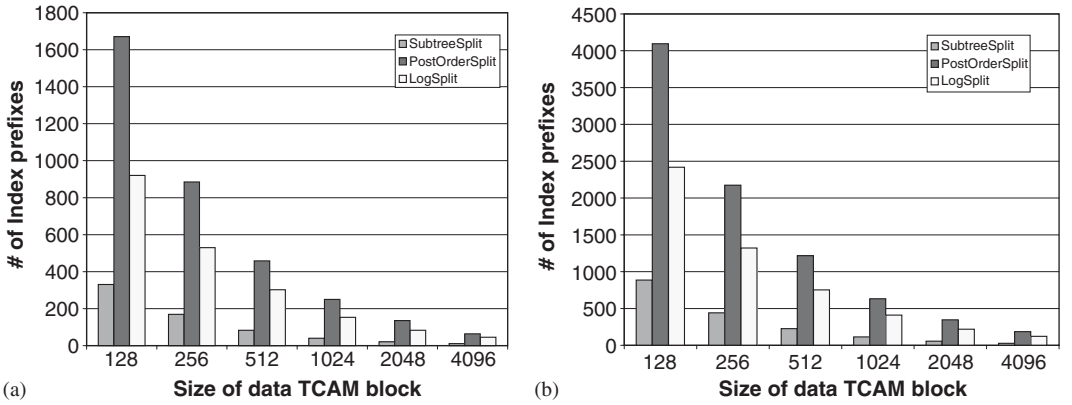

Figure 11. The number of the index prefixes *versus* the size of a data TCAM block.

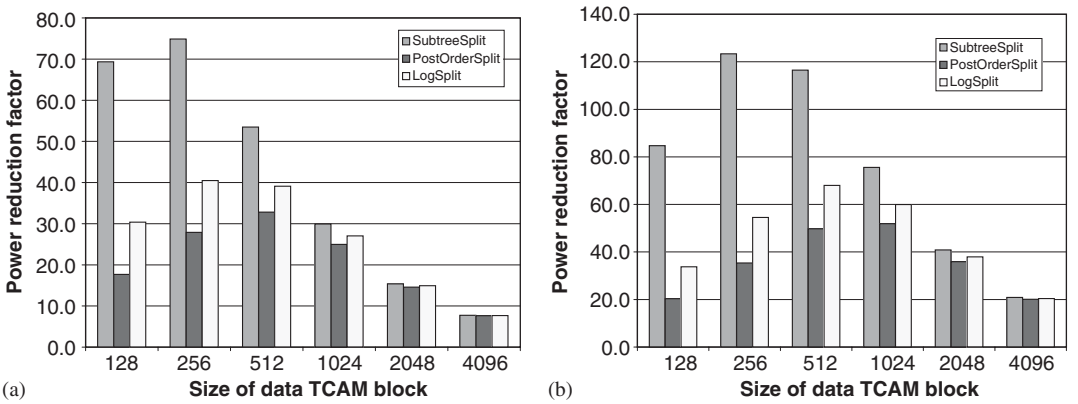

Figure 12. The power reduction factor *versus* the size of a data TCAM block: (a) AADS and (b) PAIX.

The power reduction factors are listed in Table V. The *power reduction factor* is defined as n over the sum of the number of index prefixes and m. The power consumption of the single-TCAM lookup is determined by n (the number of prefixes in the RT), since all TCAM entries are enabled for search. The power consumption of trie-based architecture (Figure 1) is determined by the size of the index TCAM and m (the size of a data TCAM block), since only the index TCAM and one data TCAM block are enabled for search. As shown in Table V, SubtreeSplit has the best power reduction ability. This comes without any surprise, since SubtreeSplit wastes data TCAM blocks to achieve small index TCAM size. The power reduction factor of LogSplit is always higher than that of PostOrderSplit. LogSplit achieves 65% for PAIX and 71% for AADS more power reduction than PostOrderSplit when $m = 128$. When the size of a data TCAM block increases, the advantage of SubtreeSplit and LogSplit over PostOrderSplit gradually decreases. This is because the power consumption of one data TCAM block starts to dominate. However, the large value of m is not desirable, since power reduction is limited when m is large. For example, the power

DOI: 10.1002/dac

Table V. The power reduction factor *versus* the size of a data TCAM block. The largest power reduction factor of each algorithm is underlined.

	m	128	256	512	1024	2048	4096
AADS	SubtreeSplit	69.3	<u>74.9</u>	53.5	29.9	15.4	7.7
	PostOrderSplit	17.7	27.9	<u>32.8</u>	25.0	14.6	7.7
	LogSplit	30.4	<u>40.5</u>	39.1	27.0	14.9	7.7
PAIX	SubtreeSplit	84.7	<u>123.4</u>	116.5	75.6	40.9	20.9
	PostOrderSplit	20.4	35.4	49.8	<u>51.9</u>	35.9	20.1
	LogSplit	33.8	54.5	<u>68.0</u>	60.0	37.9	20.4

Table VI. Worst-case performance.

	SubtreeSplit	PostOrderSplit	LogSplit
Number of data TCAM blocks	$\lceil 2n/m \rceil$	$\left\lfloor \frac{n+m}{m-W-1} \right\rfloor$	$\left\lfloor \frac{n+m}{m-\log_2 m} \right\rfloor$
Size of index TCAM	$\lceil 2n/m \rceil$	$\left\lfloor \frac{n+m}{m-W-1} \right\rfloor (W+1)$	$\left\lfloor \frac{n+m}{m-\log_2 m} \right\rfloor \log_2 m$

reduction factor of PostOrderSplit is only 7.7 for AADS when $m = 4096$, comparing with 32.8 when $m = 512$. Also, observe that the smallest m does not lead to the maximum power reduction. The largest power reduction factor of each of the three algorithms is underlined in Table V.

4.2. Performance in the worst case

It is well known that router designers have to work with the worst-case requirement for the size of the index TCAM and the overall power consumption. Current core routers are expected to handle RTs with more than one million prefixes.

Table VI gives the worst-case performance of each of these three trie-partitioning algorithms. In the worst case, SubtreeSplit uses only half of the entries in each data TCAM block. Thus, the number of data TCAM blocks needed under SubtreeSplit is $\lceil 2n/m \rceil$. Since SubtreeSplit contributes one entry per data TCAM block to the index TCAM, the size of the index TCAM is $\lceil 2n/m \rceil$. In addition, SubtreeSplit adds up to one covering prefix to each data TCAM block. When a data TCAM block is half full, it has space to accommodate the covering prefix. PostOrderSplit adds up to $W+1$ covering prefixes to each data TCAM block. Let k be the number of data TCAM blocks needed. We have $(k-1)m < k(W+1)+n$, where $(k-1)m$ is the total number of entries in the first $k-1$ data TCAM blocks, and $k(W+1)+n$ is the total number of prefixes stored at the data TCAM blocks. Hence, the worst-case number of data TCAM blocks needed under PostOrderSplit is $\lfloor (n+m)/(m-W-1) \rfloor$. Since PostOrderSplit contributes up to $W+1$ entries per data TCAM block to the index TCAM, the size of the index TCAM is $k(W+1) = \lfloor (n+m)/(m-W-1) \rfloor (W+1)$. LogSplit adds up to $\log_2 m$ covering prefixes to each data TCAM block and contributes up to $\log_2 m$ prefixes per data TCAM block to the index TCAM. We can obtain the worst-case bound using the inequality: $(k-1)m < k \log_2 m + n$. Note that the worst-case performances of SubtreeSplit and LogSplit are independent of W, while the worst-case performances of PostOrderSplit is not.

DOI: 10.1002/dac

Table VII. The worst-case number of data TCAM blocks *versus* the size of a data TCAM block.

m		512	1024	2048	4096	8192	16 384
SubtreeSplit		3907	1954	977	489	245	123
PostOrderSplit	IPv4	2088	1010	497	247	123	62
	IPv6	2612	1118	522	253	125	62
LogSplit		1978	984	491	245	123	62

Note: $n = 1\,000\,000$.

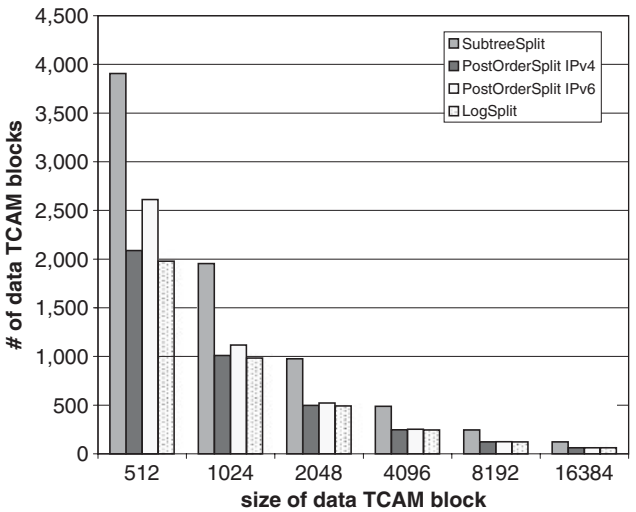

Figure 13. The worst-case number of data TCAM blocks *versus* the size of a data TCAM block.

Table VIII. The worst-case number of the index prefixes *versus* the size of a data TCAM block.

m		512	1024	2048	4096	8192	16 384
SubtreeSplit		3907	1954	977	489	245	123
PostOrderSplit	IPv4	68 904	33 330	16 401	8151	4059	2046
	IPv6	336 948	144 222	67 338	32 637	16 125	7 998
LogSplit		12 339	6820	3743	2037	1108	601

Note: $n = 1\,000\,000$.

Table VII (plotted in Figure 13) lists the worst-case number of data TCAM blocks needed, *versus* the size of a data TCAM block, when $n = 1\,000\,000$. LogSplit uses the least number of data TCAM blocks. Due to the overhead posed by covering prefixes, PostOrderSplit wastes 30% of data TCAM space when $m = 512$ and $W = 128$ (IPv6).

Table VIII (plotted in Figure 14) lists the worst-case number of the index prefixes (i.e. the size of the index TCAM), *versus* the size of a data TCAM block, when $n = 1\,000\,000$. SubtreeSplit generates the least number of index prefixes. The number of index prefixes generated by PostOrderSplit is significant, especially when $W = 128$. For example, when $m = 512$ and $W = 128$, PostOrderSplit generates up to 336 948 index prefixes, which is 34% of the RT size. But for LogSplit, the size of the index TCAM is under 1.2% of the RT size. The advantage of LogSplit

DOI: 10.1002/dac

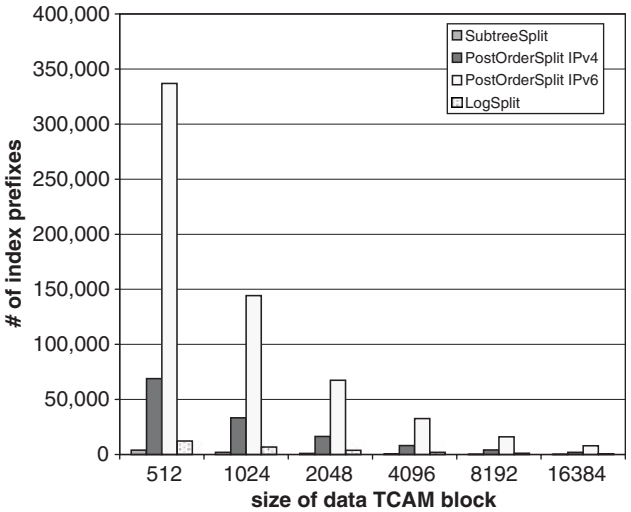

Figure 14. The worst-case number of index prefixes *versus* the size of a data TCAM block.

Table IX. The worst-case power reduction factor *versus* the size of a data TCAM block.

m		512	1024	2048	4096	8192	16 384
SubtreeSplit		226.3	<u>335.8</u>	330.6	218.1	118.5	60.6
PostOrderSplit	IPv4	14.4	29.1	54.2	<u>81.7</u>	81.6	54.3
	IPv6	3.0	6.9	14.4	27.2	<u>41.1</u>	41.0
LogSplit		77.8	127.5	<u>172.7</u>	163.1	107.5	58.9

Note: $n = 1\,000\,000$. The largest power reduction factor of each algorithm is underlined.

over PostOrderSplit is more significant when m becomes smaller. The index TCAM size of LogSplit is 18–30% of that of PostOrderSplit for IPv4, and less than 1% of that of PostOrderSplit for IPv6.

Table IX (plotted in Figure 15) lists the worst-case power reduction factor, *versus* the size of a data TCAM block, when $n = 1\,000\,000$. The power reduction factor is equal to n over the sum of m and the size of the index TCAM. As expected, the ability of PostOrderSplit to save power is severely limited by W, especially when m is small. For example, when $m = 2K$, the worst-case power reduction factor of PostOrderSplit is 54.2 for IPv4 and 14.4 for IPv6, while the worst-case power reduction factor of LogSplit is 172.7. The largest power reduction factor of PostOrderSplit is 81.7 for IPv4 and 41.1 for IPv6, while the largest power reduction factor of LogSplit is 172.7 for both IPv4 and IPv6. Though SubtreeSplit achieves the largest power reduction factor (335.8), it wastes half of the entries in data TCAM blocks.

4.3. Reduce index TCAM size further

LogSplit adds one prefix into index TCAM for each subtrie pruned. It may not be worthwhile to do so for subtries that have only a few prefixes. Besides, since RTs are frequently updated and an overflowing TCAM block requires expensive repartitioning, a few data TCAM entries should be reserved to avoid frequent overflow. Note that the size of a subtrie (i.e. the *count* value of the

DOI: 10.1002/dac

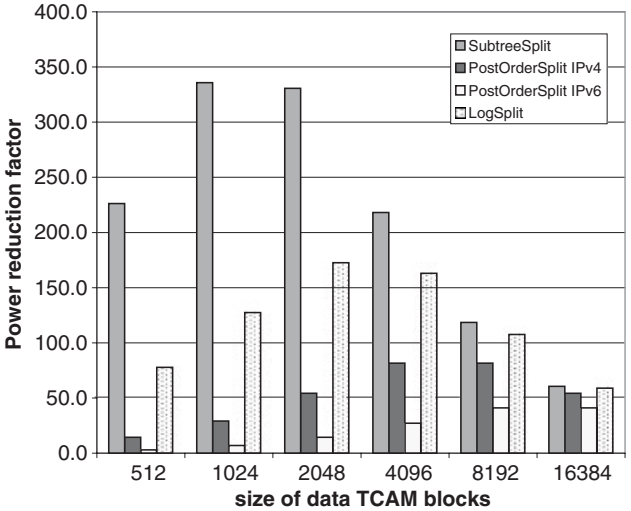

Figure 15. The worst-case power reduction factor *versus* the size of a data TCAM block.

subtrie root) pruned by LogSplit is strictly exponentially decreasing (Line 07 in Figure 6). The last four iterations (Lines 07–13 in Figure 6) find subtries containing 16, 8, 4, 2 and 1 prefixes, respectively, in the worst case. Therefore, we can reduce the worst-size number of iterations (each iteration prunes a subtrie) by four if we plan to reserve 31 empty entries per data TCAM block.

5. RELATED WORK

McAuley and Francis [1] use TCAM for RT lookup. Kobayashi *et al.* [5] associate each TCAM entry with a priority to eliminate the need to sort TCAM in the descending order of prefix length. Shah and Gupta [2] propose efficient algorithms to insert/delete a TCAM entry. Liu [6] proposes two techniques, pruning and mask extension, to compact router tables stored in TCAM. Ravikumar and Mahapatra [7] also use prefix properties to compact TCAM entries.

Panigrahy and Sharma [8] propose the idea of partitioning prefixes into equally sized groups. The partition is based on the bits selected from prefixes. The search only goes to one group in order to reduce power consumption. Zane *et al.* [3] give a greedy algorithm to select the bits and propose trie-based architecture, which is more efficient than the bit-selection based architecture. Zheng *et al.* [9] use bits 10-13 in IPv4 prefixes to partition a RT into 16 groups, and then use a greedy algorithm to assign and duplicate groups into different TCAM blocks to achieve power reduction and load balance. The scheme requires traffic statistics in order to determine duplications.

6. CONCLUSION

TCAM, which uses parallelism to achieve lookup in a single cycle, is a simple and efficient solution for RT lookup. However, TCAM has very high-power consumption because it simultaneously checks all memory entries. The trie-based architecture reduces power consumption. The idea is to use a index TCAM (always on) to select only one of many data TCAM blocks for lookup. Zane *et al.* [3] propose SubtreeSplit and PostOrderSplit to partition one-bit tries. SubtreeSplit uses

DOI: 10.1002/dac

only half of the data TCAM blocks in the worst case. Although PostOrderSplit fully utilizes each data TCAM block except the last one, its effectiveness in power reduction is limited, because it generates a large index TCAM. In the worst case, PostOrderSplit generates $W + 1$ index TCAM entries per data TCAM block, where W is 32 for IPv4 and 128 for IPv6. Since router designers have to use the worst-case bound to decide a power budget, it is necessary to reduce the worst-case size of the index TCAM.

In this paper we develop a new trie-partitioning algorithm, LogSplit, to reduce power consumption. Each data TCAM block contributes at most $\log_2 m$ entries to the index TCAM for both IPv4 and IPv6, where m is the maximum number of entries in one data TCAM block. For two real-world RTs, the number of index prefixes generated by LogSplit is 55–70% of that generated by PostOrderSplit (Table III); the largest power reduction factor of LogSplit is 41 for AADS and 68 for PAIX, while the largest power reduction factor of PostOrderSplit is 33 for AADS and 52 for PAIX (Table V). In the worst case, the index TCAM size of LogSplit is 18–30% of that of PostOrderSplit for IPv4, and less than 1% of that of PostOrderSplit for IPv6 (Table VIII); the largest power reduction factor of LogSplit is 173 for both IPv4 and IPv6, while the largest power reduction factor of PostOrderSplit is 82 for IPv4 and 41 for IPv6 (Table IX). Since the index TCAM is always enabled for search and it counts for a significant portion of the overall power consumption, especially when m is small, reducing the size of the index TCAM makes lookup engines cooler.

REFERENCES

1. McAuley A, Francis P. Fast routing table lookups using CAMs. *IEEE INFOCOM*, San Francisco, CA, U.S.A., 1993; 1382–1391.
2. Shah D, Gupta P. Fast updating algorithms for TCAMs. *IEEE Micro* 2001; **21**(1):36–47.
3. Zane F, Narlikar G, Basu A. CoolCAMs: power-efficient TCAMs for forwarding engines. *IEEE INFOCOM*, San Francisco, CA, U.S.A., 2003.
4. Merit, IPMA statistics. *http://nic.merit.edu/ipma*, 2000, 2001.
5. Kobayashi M, Murase T, Kuriyama A. A longest prefix match search engine for multi-gigabit IP processing. *Proceedings of the International Conference on Communications* (*ICC 2000*), New Orleans, U.S.A., 2000.
6. Liu H. Routing table compaction in ternary CAM. *IEEE Micro* 2002; **22**(1):58–64.
7. Ravikumar VC, Mahapatra RN. TCAM architecture for IP lookup using prefix properties. *IEEE Micro* 2004; **24**(2):60–69.
8. Panigrahy R, Sharma S. Reducing TCAM power consumption and increasing throughput. *10th Symposium on High Performance Interconnects HOT Interconnects* (*HotI'02*), Stanford University, CA, U.S.A., 2002.
9. Zheng K, Hu C, Lu H, Liu B. An ultra high throughput and power efficient TCAM-based IP lookup engine. *IEEE INFOCOM*, Hong Kong, China, 2004.

AUTHOR'S BIOGRAPHY

Haibin Lu received the BE and ME degrees in electronic engineering from Tsinghua University, Beijing, China, in 1997 and 1999, and the PhD degree in computer engineering from the University of Florida in 2003. He joined the faculty of the Department of Computer Science, University of Missouri-Columbia, as an Assistant Professor in 2003. His primary research focus lies in algorithmic aspects of computer network and multimedia communication.

DOI: 10.1002/dac