Low Power TCAMs For Very Large Forwarding Tables

Wencheng Lu and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
{wlu, sahni}@cise.ufl.edu

***Abstract*—Ternary content-addressable memories (TCAMs) may be used to obtain a simple and very fast implementation of a router's forwarding engine. The applicability of TCAMs is, however, limited by their size and high power requirement. Zane et al. [1] proposed a method and associated algorithms to reduce the power needed to search a forwarding table using a TCAM. We improve on both the algorithms proposed by them. Additionally, we show how to couple TCAMs and high bandwidth SRAMs so as to overcome both the power and size limitations of a pure TCAM forwarding engine.**

I. Introduction

Each rule of a packet forwarding table comprises a prefix and a next hop. Packet forwarding is done by determining the next hop associated with the longest prefix in the forwarding table that matches the destination address of the packet to be forwarded. Several solutions for very high-speed longest prefix matching have been proposed (see [2], [3] for surveys). Among the many proposed solutions to the packet forwarding problem, those employing TCAMs are the simplest and fastest. A TCAM is a fully associative memory in which each bit may be in one of 3 states–0, 1 and don't care. By loading the forwarding table prefixes into the TCAM in decreasing order of prefix length (ties are broken arbitrarily), the TCAM index of the longest matching prefix for any destination address may be determined in one TCAM cycle. Using this index, we can access the word of SRAM where the next hop associated with the matching prefix is stored and complete the forwarding task. So, the simplest TCAM solution to packet forwarding requires 1 TCAM search and 1 SRAM access to forward a packet. Two drawbacks of this TCAM solution are (a) an IPV4 forwarding table with n prefixes requires a TCAM that has $32n$ bits and (b) since each lookup searches the entire $32n$-bit TCAM, the power consumption is that for a TCAM of this size.

Several strategies - e.g., [4], [1], [5], [6] - have been proposed to reduce TCAM power significantly by capitalizing on a feature in contemporary TCAMs that permits one to select a portion of the entire TCAM for search. The power consumption now corresponds to that for a TCAM whose size is that of the portion that is searched. Using the example of [1], suppose we have a TCAM whose capacity is 512K prefixes and that the TCAM has a block size of 6K. So, the total number of blocks is 64. The portion of the total TCAM that is to be searched is specified using a 64-bit vector. Each bit of this vector corresponds to a block. The 1s in this vector define the portion (subtable) of the TCAM that is to be searched and the power required to search a TCAM subtable is proportional to the subtable size. While it is not required that a subtable be comprised of contiguous TCAM blocks, we assume, in this paper, that this is the case. We use the term *bucket* to refer to a set of contiguous blocks. Although, in the example of [1] the size of a bucket is a multiple of 8K prefixes, we assume that bucket sizes are required only to be integer.

This research was supported, in part, by the National Science Foundation under grant ITR-0326155

Zane et al. [1] partition the forwarding table into smaller subtables (actually, buckets) so that each lookup requires 2 searches of smaller TCAMs. Their method, however, increases the total TCAM memory that is required. Lu [6] has proposed an improved table partitioning algorithm for TCAMs. Akhbarizadeh et al. [7] propose an alternative TCAM architecture that employs multiple TCAMs and multiple TCAM selectors. The routing table is distributed over the multiple TCAMs, the selectors decide which TCAM is to be searched. The architecture of [7] is able to determine the next-hop for several packets in parallel and so achieves processing rates higher than those achievable by using a single pipeline architecture such as the one proposed by Zane et al. [1]. The proposal of Zane et al. [1], however, has the advantage that it can be implemented a commercial network processor board equipped with a TCAM and an SRAM (for example, Intel's IXP 2800 network processor supports a TCAM and up to 4 SRAMs, no customized hardware support is required) whereas that of Akhbarizadeh et al. [7] cannot.

In this paper, we improve upon the router-table partitioning algorithms of [1] and [6]. These algorithms may be used to partition router tables into fixed size blocks as is required by the architecture of [7] as well. Additionally, we show how to couple TCAMs and wide SRAMs so as to search forwarding tables whose size is much larger than the TCAM size with no loss in time and with power reduction. All of our algorithms and techniques are implementable using a commercial network processor board equipped with a TCAM and multiple SRAMs. We begin in Section II by reviewing related work. In Section III we develop an algorithm to do optimal subtree splits and in Section IV we propose a heuristic for post order split. Methods to efficiently search forwarding tables whose size is larger than the TCAM size are proposed in Sections V and VI. An experimental evaluation of the proposed methods is done in Section VII.

II. Background and Related Work

	Prefixes	Next Hop
P1	*	H1
P2	0*	H2
P3	00*	H3
P4	01*	H4
P5	11*	H5
P6	000*	H6
P7	011*	H7

Fig. 1. An example 7-prefix forwarding table

Figure 1 gives an example 7-prefix forwarding table. Figure 2 shows a simple TCAM organization for this forwarding table. In this organization, the 7 prefixes are stored in the TCAM in decreasing order of prefix length and the next hops are stored in corresponding words of an SRAM. We assume that the TCAM and SRAM words are indexed beginning at 0. Suppose that we have a packet whose destination address begins with 010. The longest matching prefix is P4. A TCAM search for the destination address returns the TCAM index 3 for the longest matching prefix. Accessing word 3 of the SRAM yields H4 as the next hop for the subject packet.

To reduce the power consumed by the TCAM search, Zane et al. [1] propose partitioning the TCAM into an index TCAM (ITCAM) and a data TCAM (DTCAM). The DTCAM is comprised of several buckets of prefixes. Each lookup requires a search of the ITCAM, a search of 1 bucket of the DTCAM, and 2 SRAM accesses. Zane et al. [1] propose two methods–subtree split and postorder split–to partition the forwarding table prefixes into DTCAM buckets. Both methods start with the 1-bit trie representation of the prefixes in the forwarding table. Figure 3 shows the 1-bit trie for the 7-prefix example of Figure 1.

In subtree split, the prefixes are partitioned into variable-size buckets. All but one of the buckets contain between $\lceil b/2 \rceil$ and b prefixes, where $b > 1$ is a specified bound on the bucket

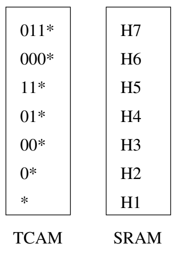

Fig. 2. Simple TCAM organization for Figure 1

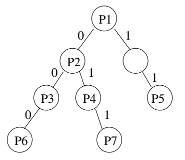

Fig. 3. 1-bit trie for 7-prefix example of Figure 1

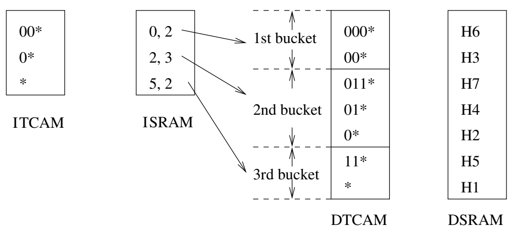

Fig. 4. 2-level TCAM organization using subtree split

size. The remaining bucket contains between 1 and b prefixes. The partitioning is accomplished by performing a postorder traversal of the 1-bit trie. During the visit operation, the subtree rooted at the visited node v is carved out if it contains at least $\lceil b/2 \rceil$ prefixes and if the subtree rooted at its parent (if any) contains more than b prefixes. The prefixes in the carved out subtree are mapped into a DTCAM bucket in decreasing order of length. A covering prefix [1] (if needed) is added to the DTCAM bucket. The covering prefix is the prefix in the nearest ancestor of v that contains a prefix. The path from the root to v defines a prefix that is added to the ITCAM. ITCAM prefixes are stored in the order they are generated by the postorder traversal. Figure 4 shows the ITCAM, DTCAM and the 2 SRAMs (ISRAM and DSRAM) for our 7-prefix example. For each ITCAM prefix, the corresponding ISRAM entry points to the start of the DTCAM bucket that generated that prefix and for each DTCAM prefix, the corresponding DSRAM entry is the next hop for that prefix. Since DTCAM buckets are of variable size, ISRAM entries will need also to store the size of the bucket pointed to. To do a lookup, the ITCAM is searched for the first prefix that matches the destination address. The corresponding ISRAM entry points to the DTCAM bucket that is to be searched next. So, by doing 2 TCAM searches and 2 SRAM accesses, we can determine the next hop for the packet.

For a forwarding table with n prefixes, the number of ITCAM entries is at most $\lceil 2n/b \rceil$ and each bucket has at most $b+1$ prefixes (including the covering prefix). Assuming that TCAM power consumption is roughly linear in the size of the TCAM being searched, the TCAM power requirement is approximately $\lceil 2n/b \rceil + b + 1$, which is minimized when $b = \sqrt{2n}$. The minimum power required is $2\sqrt{2n} + 1$. At this minimum, the total TCAM memory required is that for at most $2\sqrt{2n} + n$ prefixes (including covering prefixes; each DTCAM bucket has at most 1 covering prefix). This compares with a power and memory requirement of n for the simple TCAM solution of Figure 2. When $n = 8 * 10^4$, for example, the minimum power required by the 2-level TCAM solution of

[1]The *covering prefix* for v is the longest-length forwarding table prefix that matches all destination addresses of the form $P*$, where P is the prefix defined by the path from the root of the 1-bit trie to v. Under the assumption that $*$ is a forwarding table prefix, every v has a well-defined covering prefix. We say that the DTCAM bucket that results when the subtree rooted at v is carved out of T *needs a covering prefix* if there is a destination addresses d of the form $P*$ for which the ITCAM lookup is followed by a lookup in this DTCAM bucket and this DTCAM bucket has no matching prefix for d (equivalently, if there is no prefix on at least one downward path from v in the 1-bit trie).

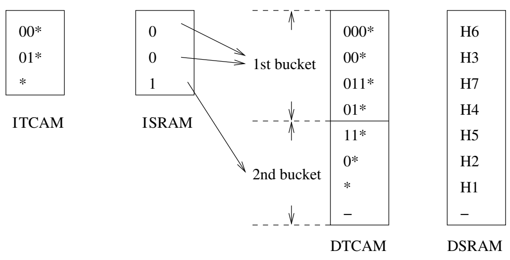

Fig. 5. 2-level TCAM organization using postorder split

Figure 4 is 801 and TCAM memory for 80,800 prefixes is required. In contrast, the simple solution of Figure 2 has a power and memory requirement of 80,000.

All but at most one of the buckets generated by postorder split [1], contain b forwarding table prefixes (plus up to W covering prefixes[2], where W is the length of the longest forwarding-table prefix); the remaining bucket has fewer than b forwarding-table prefixes (plus up to W covering prefixes). All buckets may be padded with null prefixes so that, for all practical purposes, they have the same size. The partitioning is done using a postorder traversal as in the case of subtree splitting. However, now, we may pack the prefixes of several subtrees into the same bucket so as to fill each bucket. Consequently, the ITCAM may have several prefixes for each DTCAM bucket; one prefix for each subtree that is packed into the bucket. Note also that a bucket may contain up to 1 covering prefix for each subtree packed into it. Figure 5 shows the ITCAM, ISRAM, DTCAM, and DSRAM configurations for the 7-prefix example of Figure 1.

Zane et al. [1] have shown that the size of the ITCAM is at most $(W + 1) * \lceil n/b \rceil$ and a bucket may have up to $b + W$ prefixes (including covering prefixes). Lu [6] has developed an alternative algorithm to partition into equal-size buckets. His algorithm, results in an ITCAM that has at most $\lceil n/b \log_2 b \rceil$ ITCAM prefixes and each DTCAM bucket has at most $b + \lceil \log_2 b \rceil$ prefixes (including covering prefixes); each bucket except possibly one has exactly b forwarding-table prefixes (plus up to $\lceil \log_2 b \rceil$ covering prefixes). Since $\log_2 b < W$, in practice, Lu's [6] algorithm results in smaller ITCAMs as well as reduced total space for the DTCAM. When using the partitioning algorithm of Lu [6], power is minimized when $b \approx \sqrt{n}$. At this value of b, the total TCAM memory required is that for at most $n + 1.5\sqrt{n} \log_2 n$ prefixes (including covering prefixes) and the TCAM power required is $\sqrt{n}(0.5 \log_2 n + 1) + 0.5 \log_2 n$.

In a 1-1 2-level TCAM, two levels of TCAM (ITCAM and DTCAM) are employed and each ITCAM prefix corresponds to a different DTCAM bucket. In a many-1 2-level TCAM several ITCAM prefixes may correspond to the same DTCAM bucket. Subtree splitting results in a 1-1 2-level TCAM while postorder splitting results in a many-1 2-level TCAM. In either case, a lookup requires 2 TCAM searches and 2 SRAM accesses.

[2]$W \leq 32$ for IPv4

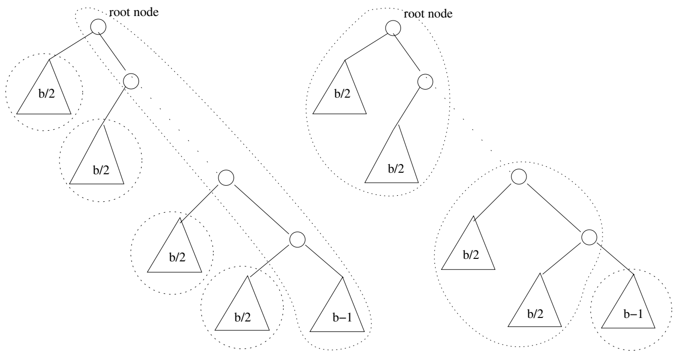

Fig. 6. Bad example for subtree split of [1]

III. Subtree Split

The subtree split algorithm of [1] is suboptimal; that is it does not partition a 1-bit trie into the smallest number of subtrees that have at most b prefixes each. In fact, the algorithm of [1] may generate almost twice the optimal number of subtrees and hence buckets and ITCAM prefixes. To see this consider the 1-bit trie of Figure 6 (a). In this, b is even, the rightmost subtrie has $b-1$ prefixes and each of the left subtries has $b/2$ prefixes. Let $h-1$ be the total number of left subtries (i.e., subtries with $b/2$ prefixes each). Figure 6 (a) shows the bucketing obtained by the algorithm of [1]. One bucket has $b-1$ prefixes and the remainder have $b/2$ prefixes each. The total number of buckets (and hence ITCAM prefixes) is h. Figure 6 (b) shows an optimal partitioning into a 1-1 2-level TCAM. The number of buckets is $h/2 + 1$. Note that since each bucket has at least $\lceil b/2 \rceil$, prefixes, 2 is an upper bound on the ratio of the number of buckets generated by the subtree split algorithm of [1] and the optimal number of buckets.

Theorem 1: Let m be the number of buckets (and hence ITCAM prefixes) generated by the subtree split algorithm of [1]. Let $m*$ be the number of buckets in an optimal subtree split. $m/m* < 2$ and this bound is best possible.

We may construct optimal subtree splits using the visit algorithm of Figure 7 in conjunction with a postorder traversal of the 1-bit trie T for the forwarding table. In the visit algorithm of Figure 7, b is the maximum number of prefixes (including the covering prefix (if any)) that may be stored in a DTCAM bucket[3], $count(x)$ is the number of prefixes stored in the subtree[4], $ST(x)$, of T that is rooted at node x of T and $split(x)$ removes $ST(x)$ from T. When $ST(x)$ is removed (split, carved) from T, the prefixes stored in $ST(x)$ together with a covering prefix for $ST(x)$ (if needed) are stored in a bucket of the DTCAM and the prefix corresponding to the path from the root of T to x is added to the ITCAM. Note that following the execution of $split(x)$, $count(r)$ decreases as $ST(r)$ has fewer nodes for every r that is an ancestor of x. Note also that whenever a subtree is split (removed, carved)

[3]The algorithm is modified easily to the case when b is the maximum number of forwarding-table prefixes that may be stored in a bucket. This is the definition of b used in [1].

[4]Note that nodes of T store only prefixes that are in the forwarding table.

```
Algorithm visit(x)
{
  if (count(x)==b){
    if (x does not require a covering prefix)
     split(x);
    else {
      // x has two children y and z.
      if (count(y)>=count(z)) split(y);
      else split(z);
    }
    return;
  }
  if (count(x)>b){
    // x has two children y and z.
    if (count(y)>=count(z)) split(y);
    else split(z);
    recompute count(x);
    if (count(x)==b)
      split(x); // x contains a prefix
  }
}
```

Fig. 7. Visit function for optimal subtree splitting

from T, the subtree contains at most b prefixes and that when the subtree contains b prefixes, no covering prefix is needed in the DTCAM bucket that results. Hence, no bucket is assigned more than b prefixes (including the covering prefix (if any)). Let $optSplit$ denote the subtree split algorithm that results from using the visit algorithm of Figure 7 in conjunction with a postorder traversal of the 1-bit trie for a forwarding table.

Theorem 2: Algorithm $optSplit$ minimizes the number of DTCAM buckets and hence minimizes the number of ITCAM prefixes.

Proof: See [8] ■

From Theorems 1 and 2, it follows that algorithm $optSplit$ results in 1-1 2-level TCAMs with the fewest number of ITCAM prefixes and up to half as many ITCAM prefixes as in the ITCAMs resulting from the algorithm of [1]. By deferring the computation of a node's $count$ until it is needed, the complexity of $optSplit$ becomes $O(nW)$, where n is the number of prefixes in the forwarding table and W is the length of the longest prefix.

The buckets created by $optSplit$ enjoy similar properties as enjoyed by those created by the subtree split algorithm of [1]. The next two theorems are similar to theorems in [1].

Theorem 3: The number of forwarding-table prefixes (this count excludes the covering prefix (if any)) in each bucket is in the range $[\lceil \frac{b}{2} \rceil$,b], except for the last bucket, which contains between 1 and b forwarding-table prefixes. When covering prefixes are accounted for, no bucket contains more than b prefixes.

Proof: Follows directly from the visit algorithm of Figure 7. Note that the buckets created by the algorithm of [1] may have up to $b+1$ prefixes (including the covering prefix). ■

Theorem 4: For a forwarding table with n prefixes, the number of DTCAM buckets generated is in the range $[\lceil \frac{n}{b} \rceil, \lceil \frac{2n}{b} \rceil]$.

Proof: Follows from Theorem 3. ■

Theorem 5: For a forwarding table with n prefixes, the power needed is that for an ITCAM search of at most $\lceil \frac{2n}{b} \rceil$ prefixes and a DTCAM search of at most b prefixes.

Proof: To search a 1-1 2-level TCAM, we search an ITCAM, a DTCAM, and also make 2 SRAM accesses. We may assume that the SRAM power is negligible. The ITCAM has as many prefixes as the number of DTCAM buckets, which by Theorem 4 is at most $\lceil \frac{2n}{b} \rceil$. Also, no DTCAM bucket has more than b prefixes. ■

IV. Postorder Split

As defined in [1], a postorder split is required to pack[5] exactly b forwarding-table prefixes into a DTCAM bucket (an unspecified number of covering prefixes may also be packed); an exception is made for 1 DTCAM bucket, which may contain up to b forwarding-table prefixes. This requirement on the number of forwarding-table prefixes per DTCAM bucket is met by packing several subtries carved from the original 1-bit trie into a single DTCAM bucket. The result is a many-1 2-level DTCAM. The algorithm of [1] may pack up to W covering prefixes into a DTCAM bucket while that of [6] packs up to $\lceil \log_2 b \rceil$ covering prefixes into a DTCAM bucket. In both algorithms, each bucket contributes a number of ITCAM entries equal to the number of carved subtrees packed into it. In this section, we propose a new algorithm for postorder split. While the variation in the number of prefixes in a bucket is the same as for the algorithm of [6] (from b to $b + \lceil \log_2 b \rceil$) and the worst-case number of ITCAM prefixes is the same for both our algorithm and that of [6], our algorithm generates much fewer ITCAM prefixes on real-world data sets. We develop also a variant of our algorithm that has the property that each DTCAM bucket other than the last one has exactly b prefixes (including covering prefixes). The last bucket may be packed with null prefixes to make it the same size as the others. When we limit each bucket to b forwarding-table prefixes, the total number of buckets is increased slightly. We use $PS1$ to refer to our postorder split algorithm that strictly adheres to the definition of [1] and we use $PS2$ to refer to the stated variant.

The strategy in $PS1$ is to first seed $\lceil n/b \rceil$ DTCAM buckets with a feasible subtree[6] of the 1-bit trie T. The size of a feasible subtree is the number of forwarding-table prefixes contained in the nodes of the subtree (this count does not include any covering prefix that may be needed by the subtree). The buckets are seeded sequentially with feasible subtrees of as large a size as possible but not exceeding b. When a feasible subtree is used to seed a bucket, the feasible subtree is carved out of T and not available for further carving[7]. Following the seeding step, we go through as many rounds of feasible tree carving and packing as needed to completely carve out T. In

[5]By packing a subtree into a DTCAM bucket, we mean that the forwarding-table prefixes in the subtree are placed into the DTCAM bucket.

[6]A *feasible subtree* of T is any subtree of T that is the result of any possible carving sequence performed on T.

[7]In general, when a feasible subtree is carved from T, we may be left with many subtrees. The feasible subtree selection process we use, however, is limited so that a single subtree remains following carving. So, the rest of our discussion assumes we have only one subtree after carving.

each round, we select the bucket B with the fewest forwarding-table prefixes. Let the number of forwarding-table prefixes in B be s. We carve from the remaining T a feasible subtree of as large a size as possible but not exceeding $b - s$ and pack this feasible subtree into B. A detailed description of PS1 and PS2 appears in [8].

V. Simple TCAM With Wide SRAM

In the simple TCAM organization of Figure 2, each word of the SRAM is used to store only a next hop. Since a next hop requires only a small number of bits (e.g., 10 bits are sufficient when the number of different next hops is up to 1024) and a word of SRAM is typically quite large (e.g., using a QDRII SRAM, we can access 72 bits (dual burst) or 144 bits (quad burst) at a time), the simple TCAM organization of Figure 2 does not optimize SRAM usage. By using each word of the SRAM to store a subtree of the 1-bit trie of a forwarding table, we can reduce the size of the required TCAM and hence reduce the power required for table lookup. The lookup time is not significantly affected as a lookup still requires 1 TCAM search (the TCAM to be searched is smaller and so the search requires less power but otherwise takes the same amount of time) and 1 SRAM access and search (the SRAM access takes the same amount of time regardless of whether a single hop or a subtree of the 1-bit trie is accessed; although the time to process the accessed SRAM word increases, the total SRAM time is dominated by the access time). To store a 1-bit subtree in an SRAM word, we use the suffix-node structure used by us in [9] to compactly store small subtrees of a 1-bit trie. Figure 8 shows this structure.

Suffix Count	len(S1)	S1	next hop of S1	···	len(Sk)	Sk	next hop of Sk	unused

Fig. 8. Suffix node format [9]

Consider a subtree of a 1-bit trie T. Let N be the root of the subtree and let $Q(N)$ be the prefix defined by the path from the root of T to N. Let $P1 \cdots Pk$ be the prefixes in the subtree plus the covering prefix for N (if needed). The suffix node for N will store a suffix count of k and for each prefix Pi, it will store the suffix Si obtained by removing the first $|Q(N)|$ bits from Pi, the length $|Si| = |Pi| - |Q(N)|$ of this suffix (the covering prefix (if any) is an exception, its suffix is $*$ and the suffix length is 0) and the next hop associated with the suffix (this is the same as the next hop associated with the prefix Pi).

Let u be the number of bits allocated to the suffix count field of a suffix node and let v be the sum of the number of bits allocated to a length field and a next-hop field. Let $len(Si)$ be the length of the suffix Si. The space needed by the suffix node fields for $S1 \cdots Sk$ is $u + kv + \sum len(Si)$ bits. Typically, we fix the size of a suffix node to equal the bandwidth (or word size) of the SRAM in use[8] and require that $u + kv + \sum len(Si)$ be less than or equal to this quantity.

[8]In some architectures, for example, it is possible to simultaneously access 1 SRAM word from each of q SRAMs. In this case, we may use a suffix node size that q times that of a single SRAM word.

In a simple TCAM with wide SRAM (referred to as STW), we carve out subtrees of the 1-bit trie for a forwarding table; each subtree is mapped into a suffix node as described above (this of course limits the size of the subtree that may be carved); and the $Q(N)$s are placed into a TCAM and the suffix nodes are placed into an SRAM in decreasing order of $Q(N)$ length.

As an example, consider the 7-prefix forwarding table of Figure 1. Suppose that a suffix node is 32 bits long (equivalently, the bandwidth of the SRAM is 32 bits). We may use 2 bits for the suffix count field (this allows up to 4 suffixes in a node as the count must be more than 0), 2 bits for the suffix length field (permitting suffixes of length up to 3), and 12 bits for a next hop (permitting up to 4096 different next hops). With this bit allocation, a suffix node may store up to 2 suffixes. Figure 9 (a) shows a carving of the 1-bit trie (Figure 3) for our 7-prefix example. This carving has the property that no subtree needs a covering prefix and each subtree may be stored in a suffix node using the stated format. Figure 9 (b) shows the STW representation for this carving.

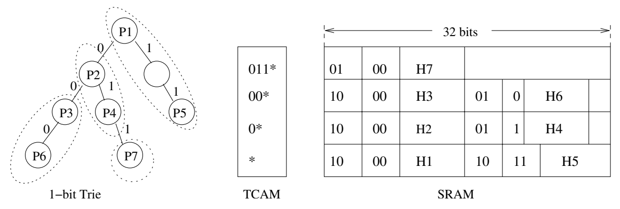

Fig. 9. Simple TCAM with SRAM (STW) for the prefix set of Figure 1

To search for the longest matching prefix (actually the next hop associated with this prefix) for the destination address d, we find first the TCAM index of the longest matching $Q(N)$ in the TCAM. This index tells us which SRAM word to search. The SRAM word is then searched for the longest suffix Si that matches d with the first $|Q(N)|$ bits stripped.

If the average number of prefixes packed into a suffix node is a_1, then the TCAM size is approximately n/a_1, where n is the total number of forwarding-table prefixes. So, the power needed for a lookup in a forwarding table using an STW is about $1/a_1$ that required when the simple TCAM organization of Figure 2 is used. Equivalently, if we have a TCAM whose capacity is n prefixes, the STW representation permits us to handle forwarding tables with up to $n * a_1$ prefixes while tables with up to only n prefixes may be handled using the organization of Figure 1; in both cases, the power and lookup time are about the same.

In the remainder of this section, we propose a heuristic to carve subtrees from T as well as a dynamic programming algorithm that does this. The heuristic attempts to minimize the number of subtrees carved (each subtree must fit in an SRAM word or suffix node) while the dynamic programming algorithm guarantees a minimum carving.

A. Carving Heuristic

Let u and v be as above and let w be the size of a suffix node. For any node x in the 1-bit trie, let $ST(x)$ be the subtree rooted at x. Note that $ST(x)$ changes as we carve out subtrees from T. Let $ST(x).numP$ be the number of prefixes in $ST(x)$ (the covering prefix (if needed) is excluded) and let $ST(x).numB$ be the number of bits needed to store the suffix lengths, suffixes and next hops for these prefixes of $ST(x)$. Clearly, when x is null, $ST(x).numP$ = $ST(x).numB$ = 0. When x is not null, let l and r be its two children (either or both may be null). We obtain the following recurrence for $ST(x).numB$.

$$ST(x).numB = \begin{cases} ST(l).numB + ST(l).numP + ST(r).numB + \\ \quad ST(r).numP + v \quad x \text{ contains a prefix} \\ ST(l).numB + ST(l).numP + ST(r).numB + \\ \quad ST(r).numP \quad \text{otherwise} \end{cases} \tag{1}$$

To see the correctness of this recurrence, notice that each prefix in $ST(l)$ and $ST(r)$ has a suffix that is 1 longer in $ST(x)$ than in $ST(l)$ and $ST(r)$. So, we need $ST(l).numB+$ $ST(l).numP + ST(r).numB + ST(r).numP$ bits to store their lengths, suffixes, and next hops. Additionally, when x contains a prefix, we need v bits to store the length (0) of its suffix as well as its next hop; no bits are needed for the suffix itself (as the suffix is $*$ and has length 0).

The size, $ST(x).size$, of the suffix node needed by $ST(x)$ is given by

$$ST(x).size = \begin{cases} ST(x).numB + u & \text{no covering prefix is needed for } x \\ ST(x).numB + u + v & \text{otherwise} \end{cases} \tag{2}$$

The correctness of Equation 2 follows from the observation that in either case, we need u additional bits for the suffix count. When a covering prefix is needed, we require also v bits for the length (which is 0) and next-hop fields for this covering prefix.

Our carving heuristic performs a postorder traversal of the 1-bit trie T (a detailed development appears in [8]). Whenever a subtree is split from the 1-bit trie, the prefixes in that subtree as well as a covering prefix (if needed) are put into a suffix node and a TCAM entry for this suffix node generated.

The overall complexity of our tree carving heuristic is $O(nW)$, where n is the number of prefixes in the forwarding table and W is the length of the longest prefix.

B. Dynamic Programming Carving Algorithm

Define a *partial subtree*, $PT(N)$, to be a feasible subtree of T that is rooted at N. Let $opt(N, b, p)$ be the minimum number of suffix nodes in any carving of $ST(N)$ under the following constraints:

1) When all but one of the subtrees represented by the suffix nodes are carved out of $ST(x)$, we are left with a partial subtree $PT(N)$. Note that since every suffix node contains at least 1 forwarding-table prefix, every carved subtree (other than $PT(N)$) contains at least 1 forwarding-table prefix.
2) $PT(N).numB = b$ and $PT(N).numP = p$.

Note that $opt(N, b, p)$ includes the suffix node needed for $PT(N)$ when $p > 0$; when $p = 0$, no suffix node is needed for $PT(n)$; and $opt(N, 0, 0) = \infty$ when N contains a forwarding-table prefix as, in this case, it is not possible to have a $PT(N)$ that contains no forwarding-table prefixes. We define $opt(N, s, y) = \infty$ whenever $s < 0$ or $y < 0$.

Let $opt(N)$ be the minimum number of suffix nodes in any carving of $ST(N)$. We develop recurrence equations from which $opt(root(T))$, the minimum number of suffix nodes in any carving of T, may be computed. In the following, $pMax$ denotes the maximum number of suffixes that may be packed into a suffix node (notice that $pMax < w/v$ and is also no more than the maximum permissible value for suffix count).

Consider an optimal carving of $ST(N)$. If $ST(N)$ needs no covering prefix, then $PT(N)$ has between 0 and $pMax$ prefixes. When a covering prefix is needed, $PT(N)$ has between 1 and $pMax - 1$ prefixes as we need space in the corresponding suffix node for the covering prefix. So,

$$opt(N) = \begin{cases} \min_{0 \le b \le w-u, 0 \le p \le pMax}\{opt(N, b, p)\} \\ \quad \text{no covering prefix is needed for } N \\ \min_{v \le b \le w-u-v, 1 \le p \le pMax-1}\{opt(N, b, p)\} \\ \quad \text{otherwise} \end{cases} \tag{3}$$

A detailed development of $opt(N, b, p)$ appears in [8].

The time to compute $opt(root(T))$ is dominated by the time to compute $opt(*, *, *)$, $O(w * pMax)$. Since $O(nWwpMax)$ $opt(*, *, *)$ are to be computed, the time required to determine $opt(root(T))$ is $O(nWw^2pMax^2) = O(nWw^4/v^2)$ (as $pMax < w/v$).

VI. 2-Level TCAM With Wide SRAM

Using the STW strategy of Section V, the power needed to search a forwarding table is approximately $1/a_1$ that required when the simple TCAM strategy of Figure 2 is used, where a_1 is the average number of prefixes packed into a suffix node. Equivalently, with the same power budget or TCAM capacity, we can handle forwarding tables that are a_1 times as large. Further gains in power reduction and increase in forwarding-table size that may be supported can be achieved by adopting a 2-level TCAM structure (ITCAM and DTCAM). Some of the possible 2-level TCAM structures that use wide SRAMs are discussed in the remainder of this section.

A. 1-1 2-Level TCAM

We consider 4 possible organizations for a 1-1 2-level TCAM with wide SRAM. The first of these (Figure 10) uses a single wide SRAM. We start with the 1-bit trie for the forwarding table and create suffix nodes as in Section V. Let U be the 1-bit trie for the $Q(N)$s stored in the TCAM for the corresponding STW organization. We apply our subtree split algorithm of Section III to carve U into DTCAM buckets of size b (each bucket has up to b prefixes (including a covering prefix if needed) of U). The ITCAM is set up as in Section III. However, for the DTCAM, we pad DTCAM buckets that have fewer than b prefixes with null prefixes. The DTCAM buckets are placed in the DTCAM in the same order as used for their corresponding ITCAM indexes; the suffix nodes are placed into wide SRAM so that the suffix node in the ith SRAM word corresponds to the prefix in the ith DTCAM position. The defined 1-1 2-level TCAM organization is referred to as the 1-12Wa organization. Figure 10 shows the layout for the 7-prefix forwarding-table example of Figure 1. This layout uses DTCAM buckets with $b = 3$.

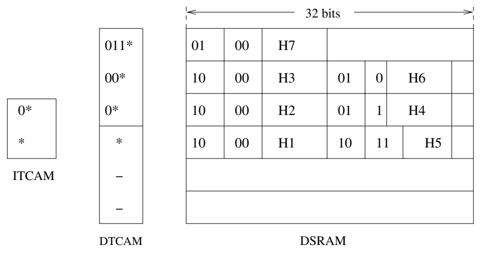

Fig. 10. 1-12Wa with fixed-size DTCAM buckets

To search for the longest matching prefix of d using the 1-12Wa organization, we first search the ITCAM for the first ITCAM entry that matches d. From the index of this ITCAM entry and the DTCAM bucket size b, we compute the location of the DTCAM bucket that is to be searched. The identified DTCAM bucket is next searched for the first entry that matches d. The SRAM word corresponding to this matching entry is then searched for the longest matching prefix using the search strategy for a suffix node. In all, 2 TCAM searches and 1 SRAM search are done. The power reduction, relative to the STW organization, is by a factor equal that provided by the subtree split scheme of Section III (the reduction factor is approximately $n/(a_1 b)$). Additionally, the number of SRAM accesses is only 1 vs 2 for the scheme of Section III. However, 1-12Wa may waste up to half of the DTCAM because the subtree split algorithm of Section III may populate DTCAM buckets with as few as $\lceil b/2 \rceil$ prefixes.

We can overcome the problem of inefficient DTCAM space utilization by 1-12Wa by introducing an ISRAM (this may just be a logical partition of the SRAM used for suffix nodes) as is done in a 2-level TCAM organization that uses subtree split (Figure 4). Now, following the search of the ITCAM, an ISRAM access is made to determine the start of the DTCAM bucket that is to be searched. This variant of 112Wa is referred to as 1-12Wb. Figure 11 shows the 1-12Wb layout for our 7-prefix example.

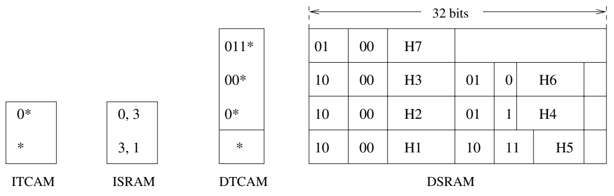

Fig. 11. 1-12Wb with variable-size DTCAM buckets

Two additional organizations, 1-12Wc and 1-12Wd result from recognizing that the ISRAM could be used to store a suffix node rather than just a pointer to a DTCAM bucket. 1-12Wc (Figure 12) uses the fixed DTCAM bucket size organization used by 1-12Wa while 1-12Wd uses the variable DTCAM bucket organization of 1-12Wb. The suffix nodes in the ISRAM are constructed from the 1-bit trie V for the prefixes used in the ITCAM of Figures 10 and 11. This construction of suffix nodes uses one of the algorithms given in Section V. The prefixes in the ITCAM for the 1-12Wc and 1-12Wd organizations correspond to those for its ISRAM suffix nodes.

To search using 1-12Wc, for example, we first search the ITCAM for the first entry that matches d, then the corresponding suffix node in the ISRAM is accessed and searched using the search method for a suffix node. This search yields the same result as obtained by searching the ITCAM of the 1-12Wa representation. Since DTCAM buckets are of a fixed size, using the single pointer stored in the searched ISRAM suffix node, we can determine which DTCAM bucket to search next.

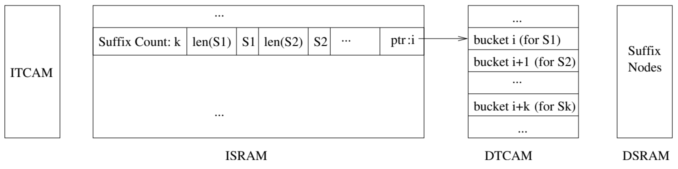

Fig. 12. 1-12Wc with fixed-size DTCAM buckets

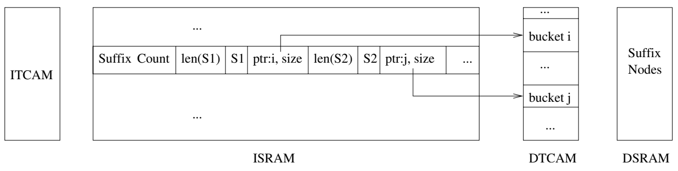

Fig. 13. 1-2Wd with variable-size DTCAM buckets

B. Many-to-one 2-Level TCAM

The many-1 2-level TCAM with wide memory (M-12W) uses fixed-size DTCAM buckets that are filled to capacity with prefixes from U using the postorder split algorithm of Section IV. Two variants (M-12Wa and M-12Wb, see Figure 14) are possible depending on whether the ISRAM simply stores pointers to DTCAM buckets (as in Figure 5) or it stores suffix nodes formed from V.

The search process for an M-12Wa (b) is the same as that for a 1-12Wb (d).

VII. Experimental Results

C++ codes for our algorithms were compiled using the GCC 3.3.5 compiler with optimization level `O3` and run on a 2.80 GHz Pentium 4 PC. We compared the performance of our algorithms with that of recently published algorithms [1], [6] to construct low-power 2-level TCAMs for very large forwarding tables. For our wide SRAM strategies, we assume a QDRII SRAM (quad burst) that supports the retrieval

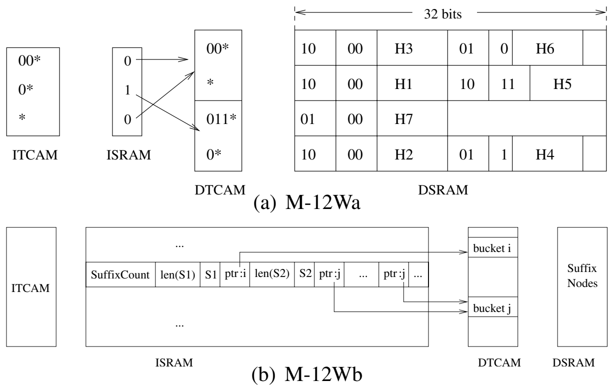

Fig. 14. Many-to-one 2-level TCAM with wide SRAM

of 144 bits of data with a single memory access. For all implementations, we allocate 12 bits for each next hop field. For the ISRAM in 1-12Wb and 1-12Wd, the size of the pointer pointing to a DTCAM entry was 16 bits and another 10 bits were used to specify the actual size of a bucket. For the ISRAM in 1-12Wc, M-12Wa and M-12Wb, the size of the pointer pointing to a DTCAM bucket was 10 bits.

A. *IPv4 Router Tables*

For the IPv4 tests, we used three IPv4 router tables AS1221, AS3333, and AS4637 that were downloaded from [10]. The number of prefixes in these router tables is 281516, 211968 and 210119, respectively.

1) 2-level TCAMs Without Wide SRAMs: First, we compared our 1-1 2-level TCAM algorithm *optSplit* with the corresponding algorithm *subtree-split* of [1]. Recall that for any given DTCAM bucket size, *optSplit* results in an ITCAM of minimum size, where size of a TCAM is the number of TCAM entries. Note also that, for 1-1 2-level TCAMs, the ITCAM size equals the number of DTCAM buckets. Even though *subtree-split* may generate ITCAMs whose size is up to twice optimal, on our 3 IPv4 test sets, the ITCAMs generated by *subtree-split* were only between 1.9% and 3.4% larger than optimal; the average and standard deviation were 2.9% and 0.1%, respectively.

For many-1 2-level TCAMs, we compared our algorithms $PS1$ and $PS2$ with $postorder-split$ of [1] and $triePartition$ of [6]. Though [6] has established the superiority of $triePartition$ to $postorder-split$ in the worst case analysis, it didn't compare them in terms of real-life router tables. Figure 15 plots the ITCAM size of DTCAM buckets constructed by these three algorithms for AS1221. We see that $PS2$ has the best performance. The ITCAMs constructed by $triePartition$ are from 80% to 137% larger than those constructed by $PS2$ with the average and standard deviation being 98% and 48%, respectively. The size of the ITCAMs constructed by $PS1$ were between 0.94 and 1.22 times that of the ITCAMs constructed by $PS2$; the average and standard deviation were 1.08 and 0.16, respectively. Between $triePartition$ and $posorder-split$, $posorder-split$ required 29% to 38% larger ITCAMs with 34% in average and 3% as the standard deviation. The number of DTCAM buckets constructed by $triePartition$ was between 4% to 7% more than that constructed by $PS2$; the average and standard deviation being 3% and 1%, respectively. $PS1$ and $triePartion$ resulted in the same number of DTCAM buckets as did $postorder-split$.

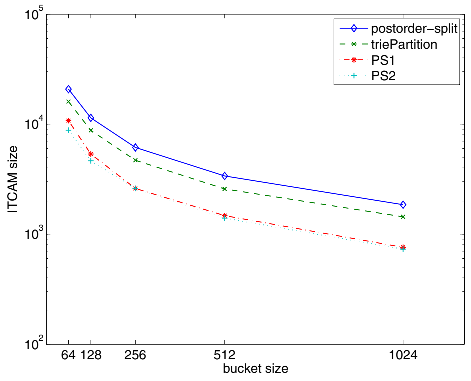

Fig. 15. ITCAM size for many-1 2-level TCAMs for AS1221.

2) 2-level TCAMs With Wide SRAMs: For the benchmarking of 2-level TCAMs with wide SRAMs, we used $optSplit$ for 1-1 2-level TCAMs and $PS2$ for many-1 2-level TCAMs. Our experiments indicated that the carving heuristic and the dynamic programming carving algorithm of Section V give similar results. Since the heuristic is considerably faster, we used the carving heuristic for benchmarking here. Figure 16 plots the total TCAM size (ITCAM plus DTCAM) constructed by each of our 6 wide-SRAM algorithms (1-12Wa, 1-12Wb, 1-12Wc, 1-12Wd, M-12Wa, and M-12Wb) for AS1221.

The six strategies cluster into two groups 1-12Wa and 1-12Wc being the first group and the remaining 4 defining the second group. The TCAM size is about the same for each strategy in the same group. Strategies in the first group required between 26% to 35% more TCAM memory than required by strategies in the second group.

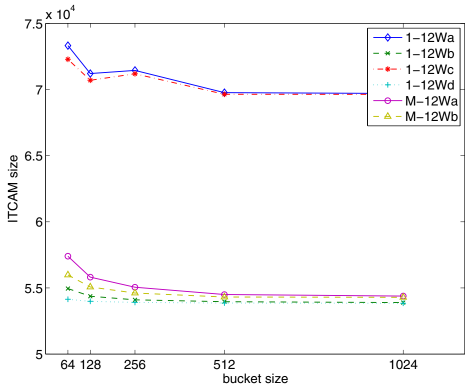

Fig. 16. Total TCAM size with wide SRAMs for AS1221.

Figure 17 plots the total TCAM power required by our 6 strategies for AS1221. On the power metric, 1-12Wc is the clear winner.

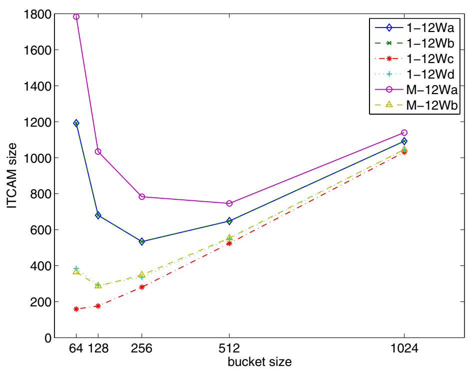

Fig. 17. Total TCAM power with wide SRAMs for AS1221.

Figure 18 plots the total SRAM size required by our 6 strategies for AS1221. The strategies cluster into two groups with strategies in the same group requiring about the same amount of SRAM. The first group comprises 1-12Wb, 1-12Wd, M-12Wa, and M-12Wb while the second group comprises 1-12Wa and 1-12Wc. The SRAM requirement of strategies in the first group are between 26% to 35% larger than that for those in the second group; the average being 29%.

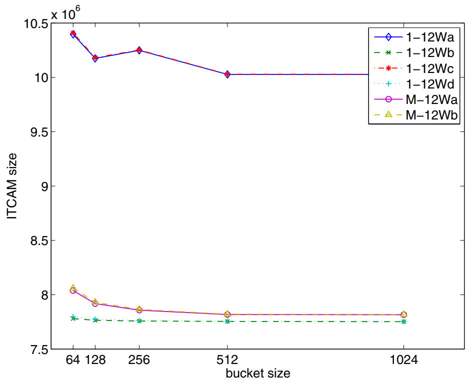

Fig. 18. Total SRAM size with wide SRAMs for AS1221.

3) 2-level TCAMs Without Wide SRAMs Vs. 2-level TCAMs With Wide SRAMs: Now, we compare the best two algorithms for 2-level TCAMs without wide SRAMS, $optSplit$ and $PS2$, with the two best strategies for 2-level TCAMs with wide SRAMs, 1-12Wc and M-12Wb. The total TCAM size, total TCAM power, and total SRAM size for each of 3 data sets using these four algorithms are reported in [8]. In terms of total TCAM size and TCAM power, 1-12Wc and M-12Wb are significantly superior to $optSplit$ and $PS2$. Both $optSplit$ and $PS2$ required more than 5 times the TCAM required by M-12Wb; $optSplit$ also required more than 6 times as much TCAM power, and $PS2$ required about 10 times as much TCAM power as required by the strategies employing wide SRAM. $optSplit$ required slightly smaller total TCAM than $PS2$, and much less total TCAM power than $PS2$; both require about the same amount of SRAM. Both $optSplit$ and $PS2$ require about 66% less SRAM than required by 1-12Wc and about 56% less SRAM than required by M-12Wb. Since TCAM is more expensive than SRAM and also consumes more power, we recommend 1-12Wc and M-12Wb over $optSplit$ and $PS2$.

VIII. Conclusion

We have developed an optimal algorithm, $optSplit$, for subtree splitting and shown that, in the worst case, this algorithm may generate half as many ITCAM entries (equivalently, DTCAM buckets) when partitioning a 1-bit trie as generated by the heuristic, $subtree - split$ of [1]. However, on our test data, the heuristic of [1] generated near-optimal partitions. For many-1 partitioning, our heuristic $PS2$ outperforms the heuristic $triePartition$ of [6]. In fact, on IPv4 data, $triePartition$ results in 80% to 137% more ITCAM entries than generated by $PS2$ on our test data.

Besides improving upon existing trie partitioning algorithms for TCAMs, we have proposed a novel way to combine TCAMs and SRAMs so as to achieve a significant reduction in power and TCAM size. This is done without any increase in the number of TCAM searches and SRAM accesses required by a table lookup! Note that regardless of whether we use the many-1 2-level schemes of [1], [6] or the recommended wide memory schemes M-12Wb and 1-12Wc developed by us, a lookup requires 2 TCAM searches and 2 SRAM accesses. However, on our IPv4 test data, M-12Wb required about 1/5th the TCAM memory and about 1/10 the TCAM power as required by our improved versions of the schemes of [1], [6]; however, M-12Wb required 2.5 times as much SRAM memory. On IPv4 data, 1-12Wc required about 1/4th the TCAM memory, 1/12th as much TCAM power, and about 3 times as much SRAM memory as required by our improved versions of the schemes of [1], [6]. Since TCAM memory and power are the dominant criteria for optimization, we recommend M-12Wb when we wish to optimize TCAM memory and 1-12Wc when we wish to optimize power.

References

[1] G. N. F. Zane and A. Basu, “CoolCAMs: Power-Efficient TCAMs for Forwarding Engines,” in *INFOCOM*, 2003.
[2] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, “Survey and Taxonomy of IP Address Lookup Algorithms,” *IEEE Network*, 2001.
[3] S. Sahni, K. Kim, and H. Lu, “Data Structures for One-dimensional Packet Classification using Most-specific-rule Matching,” *International Journal on Foundations of Computer Science*, vol. 14, no. 3, 2003.
[4] R. Panigrahy and S. Sharma, “Reducing TCAM power consumption and increasing throughput,” in *IEEE Symposium on High Performance Interconnects Hot Interconnects*, 2002.
[5] H. L. K. Zheng, C. Hu and B. Liu, “An ultra high throughput and power efficient TCAM-based IP lookup engine,” in *IEEE INFOCOM*, 2004.
[6] H. Lu, “Improved Trie Partitioning for Cooler TCAMs,” in *IASTED International Conference on Advances in Computer Science and Technology*, 2004.
[7] R. P. M. Akhbarizadeh, M. Nourani and S. Sharma, “A TCAM-based Parallel Architecture for High-speed Packet Forwarding,” *IEEE Transactions on Computers*, vol. 56, no. 1, 2007.
[8] W. Lu and S. Sahni. (2007) Low Power TCAMs For Very Large Forwarding Tables. University of Florida. [Online]. Available: http://www.cise.ufl.edu/~wlu/papers/tcam.pdf
[9] ——, “Succinct Representation of Static Packet Classifiers,” in *International Conference on Computer Networking*, 2007.
[10] [Online]. Available: http://bgp.potaroo.net