CoolCAMs: Power-Efficient TCAMs for Forwarding Engines

Francis Zane Girija Narlikar Anindya Basu
Bell Laboratories, Lucent Technologies
700 Mountain Ave, Murray Hill, NJ 07974
{francis,girija,basu}@research.bell-labs.com

Abstract— **Ternary Content-Addressable Memories (TCAMs) are becoming very popular for designing high-throughput forwarding engines on routers: they are fast, cost-effective and simple to manage. However, a major drawback of TCAMs is their high power consumption. This paper presents architectures and algorithms for making TCAM-based routing tables more power efficient. The proposed architectures and algorithms are simple to implement, use commodity TCAMs, and provide worst-case power consumption guarantees (independent of routing table contents).**

I. Introduction

Ternary Content Addressable Memories (TCAMs) are fully associative memories that allow a "don't care" state to be stored in each memory cell in addition to 0s and 1s. This feature makes them particularly attractive for packet classification and route lookup applications which require longest prefix matches. When a destination address is presented to the TCAM, each TCAM entry is looked up in parallel, and the longest prefix that matches the address is returned. Thus, a single TCAM access is sufficient to perform a route lookup operation. In contrast, conventional ASIC-based designs that use tries may require multiple memory accesses for a single route lookup. Therefore, routing latencies for TCAM-based routing tables are significantly lower than ASIC-based tables. Moreover, TCAM-based tables are typically much easier to manage and update than tables implemented using tries.

Despite these advantages, routing vendors have been slow in adopting TCAM devices in packet forwarding engines because of two main reasons. First, TCAM devices have traditionally been more expensive and less dense compared to conventional ASIC-based devices. However, both the density and the cost of TCAMs have dramatically improved in the past few years, making them a viable alternative to ASIC-based designs in high-speed core routers. The second reason is that of high power consumption. Current high-density TCAM devices consume as much as 12–15 Watts each when all the entries are enabled for search. Moreover, a single linecard may require multiple TCAMs to handle filtering and classification as well as IP lookup on large forwarding tables. This high power consumption number affects costs in two ways—first, it increases power supply and cooling costs that account for a significant portion of an ISP's operational expenses [1]. Second, it reduces port density since higher power consumption implies that fewer ports can be packed into the same space (e.g., router rack) due to cooling constraints. Therefore, it is important to minimize the power budget for TCAM-based forwarding engines to make them economically viable.

In this paper, we focus on the problem of making TCAM-based forwarding engines more power efficient by exploiting commonly available TCAM features. Several TCAM vendors (e.g., [3]) now provide mechanisms for searching only a part of the TCAM device in order to reduce power consumption during a lookup operation. We take advantage of this feature to provide two different power efficient TCAM-based architectures for IP lookup. Both of our architectures utilize a two stage lookup process. The basic idea in either case is to divide the TCAM device into multiple partitions (depending on the power budget). When a route lookup is performed, the results of the first stage lookup are used to selectively search only one of these partitions during the second stage lookup. The two architectures differ in the mechanism for performing the first stage lookup. In the first architecture, we use a subset of the destination address bits to hash to a TCAM partition (the *bit-selection* architecture), allowing for a very simple hardware implementation. The selected bits are fixed based on the contents of the routing table. In the second architecture, a small trie (implemented using a separate, small TCAM) is used to map a prefix of the destination address to one of the TCAM partitions in the next stage (the *trie-based* architecture). This adds some design complexity, but we show that it results in significantly better worst-case power consumption.

The main contributions of this paper are as follows. First, for each architecture, we bound the worst-case power consumption. For the bit-selection architecture, the bounds will depend on some assumptions regarding prefix distributions. The worst-case bounds provide hardware designers with a worst-case power budget. Second, we present partitioning algorithms for both architectures, and analyze the performance of these algorithms. Finally, we evaluate our partitioning algorithms using real routing table traces obtained from various core routers in the Internet. We show that in realistic settings, the power savings are larger than are guaranteed by the worst-case analysis. We also note that while this paper focuses on IPv4 address lookups, similar techniques can be used for IPv6 address lookups.

The rest of the paper is organized as follows. Section II describes the architecture of a typical TCAM device. Section III describes the bit-selection architecture along with an algorithm with good average case performance as well as a bound on the worst-case power budget. Section IV describes the trie-based architecture, along with two algorithms and their bounds on the worst-case power budgets. In both these sections, we validate our results using real life routing table traces. Section V addresses issues involved in updating the routing tables on both the proposed architectures. Related work is summarized in Section VI, and we conclude in Section VII.

II. TCAMs for address lookups

A Ternary Content Addressable Memory (TCAM) is a fully associative memory that allows a "don't care" state for each memory cell, in addition to a 0 and a 1. A memory cell in a "don't care" state matches both 0s and 1s in the corresponding input bit. The contents of a TCAM can be searched in parallel and a matching entry, if it exists, can be found in a single cycle (using a single TCAM access). If multiple entries match the input, the entry with the lowest address in the TCAM is typically returned as the result.

The characteristics described above make TCAMs an attractive technology for IP route lookup operations where the destination address of an incoming packet is matched with the *longest matching prefix* in a routing table database. TCAMs can be used to implement routing table lookups as follows. If the maximum prefix length is W, then each routing prefix of length $n(\leq W)$ is stored in the TCAM with the rightmost $W-n$ bits as "don't cares". For example, the IPv4 prefix 192.168.0.0/15 will have "don't care" in the last 17 bit positions. To ensure that the longest prefix match is returned, the prefixes in the TCAM must be sorted in order of decreasing prefix length. The sorting requirement makes it difficult to update the routing table—however, recent work [9] has proposed innovative algorithms for performing TCAM updates simply and efficiently.

As mentioned earlier, the two main disadvantages of using TCAMs have traditionally been the high cost to density ratio and the high power consumption. Recent developments in TCAM technology have effectively addressed the first issue—TCAM devices with high capacity (up to 18Mbits) and search rates of over 100 Million lookups/second [3], [8] are now coming to market with costs that are competitive with alternative technologies (such as pipelined ASIC-based routing engines).

The power consumption issue still remains somewhat unresolved. The main component of power consumption in TCAMs is proportional to the number of searched entries. A typical 18Mbit TCAM device can consume up to 15 Watts of power when all the entries are searched. Growth trends in the routing tables in the Internet core [2] have prompted routing vendors to design routing engines capable of scaling up to 1 million entries. A 18Mbit TCAM can store up to 512K 32 bit prefixes—this translates to at least 2 TCAM devices for IPv4 forwarding alone. Adding more TCAM devices for flow classification and IPv6, one can see how TCAM power consumption on a linecard can become a major cost overhead.

TCAM vendors today have started providing mechanisms that can reduce power consumption by selectively addressing smaller portions of the TCAM. Each portion (called a sub-table or database) is defined as a set of TCAM blocks. A TCAM block is a contiguous, fixed-sized chunk of TCAM entries, usually much smaller than the size of the entire TCAM. For example, a 512K entry TCAM could be divided into 64 blocks containing 8K entries each. The sub-tables can then be defined as (possibly overlapping) subsets of the 64 blocks by using a 64-bit mask. When a search command is issued, the sub-table ID is also specified along with the input—only the blocks in the specified sub-table are then searched. Currently, TCAMs typically support a small number of sub-tables (such as 8 sub-tables addressed by a 3-bit ID), but the same mechanism could be used to support more sub-tables. Typically, each sub-table is intended for use in a different lookup/classification application (e.g., IPv4 lookup, IPv6 lookup, flow classification, and so on).

In this paper, we exploit the mechanism described above to reduce power consumption for route lookup applications. Given that the power consumption of a TCAM is linearly proportional to the number of searched entries, we use this number as a measure of the power consumed. Clearly, if the TCAM is partitioned into K equal-sized sub-tables, it is possible to reduce the maximum number of entries searched per lookup operation to as low as $\frac{1}{K}$ of the TCAM size. However, this raises three important issues. First, we need to partition the TCAM into sub-tables. Second, given an input, we need to select the right partition and search it. Finally, for a given partitioning scheme, we need to compute the size of the largest partition over all possible routing tables (worst-case bound) so that hardware designers can allocate a power budget. We now examine how these issues can be addressed by presenting two different architectures in the next two sections.

III. The Bit Selection Architecture

In this section, we describe the bit selection architecture for TCAM-based packet forwarding engines. The core idea here is to split the entire routing table stored in the TCAM device into multiple sub-tables or *buckets*, where each bucket is laid out over one or more TCAM blocks. Each route lookup is now a two-stage operation where a fixed set of bits in the input is used to hash to one of the buckets. The selected bucket is then searched in the second stage. The hashing is performed by some simple glue logic placed in front of the TCAM device (which we refer to as the *data TCAM*). We restrict the hash function here to be such that it simply uses the selected set of input bits (called the *hashing bits*) as an index to the appropriate TCAM bucket.

In the following subsections, we first describe the basic architecture for the forwarding engine, followed by a data-independent bound on the worst-case power consumption. This bound is dependent on the size of the routing table and is proportional to the maximum number of blocks searched for any lookup. We then describe some heuristics to efficiently split a given routing table into buckets and how to map these buckets onto TCAM blocks. Finally, we describe some experimental results obtained by applying our heuristics on real routing tables. The problem of updating the routing table is addressed in Section V.

A. Forwarding engine architecture

The forwarding engine design for the bit selection architecture is based on a key observation made in a recent study [2] of routing tables in the Internet core. This study pointed out that a very small percentage of the prefixes in the core routing tables (less than 2% in our datasets) are either very short (<16 bits) or very long (>24 bits). We therefore developed an architecture where the very short and very long prefixes are grouped into the minimum possible number of TCAM blocks. These blocks are searched for every lookup. The remaining 98% of the prefixes that are 16 to 24 bits long are partitioned into buckets, one of which is selected by hashing for every lookup.

The bit-selection architecture is shown in Figure 1. The TCAM blocks containing the very short and very long prefixes are not shown explicitly. The bit-selection logic in front of the TCAM is a set of muxes that can be programmed to extract the hashing bits from the incoming packet header and use them to index to the appropriate TCAM bucket. The set of hashing bits can be changed over time by reprogramming the muxes.

For the rest of this section, we make the following assumptions. First, we only consider the set of 16 to 24 bit long prefixes (called the *split set*) for partitioning. Second, it is possible that the routing table will span multiple TCAM devices, which would then be attached in parallel to the bit selection logic. However, each lookup would still require searching a bucket in a single TCAM device. Thus, for simplicity, we assume that there is only one TCAM device. Third, we assume that the total number of buckets $K=2^k$ is a power of 2. Then, the bit selection logic extracts a set of k hashing bits from the packet header and selects a prefix bucket. This bucket, along with

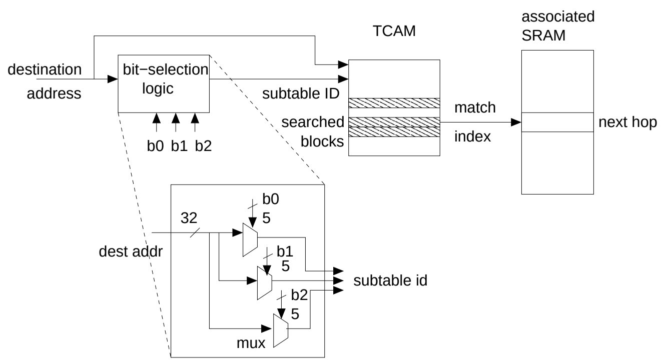


Fig. 1. Forwarding engine architecture for using bit selection to reduce power consumption. The 3 hashing bits here are selected from the 32-bit destination address by setting the appropriate 5-bit values for b_0, b_1 and b_2.

the TCAM blocks containing the very short and very long prefixes are then searched.

The two main issues now are how to select these k hashing bits, and how to allocate the different buckets among the various TCAM blocks. (Recall that the bucket size may not be an integral multiple of a TCAM block size.) The first issue leads to our final assumption—we restrict ourselves to choosing the hashing bits from the first 16 bits, which is the minimum length of a prefix in the split set. Otherwise, if k' (say) of the hashing bits are in bit positions longer than the length of a prefix, the prefix needs to be replicated in $2^{k'}$ buckets. The “best” hash function (that is, set of hashing bits) is the one that minimizes the size of the biggest resulting bucket.

B. *Worst-case power consumption*

We begin by analyzing the worst-case power consumption of the bit-selection architecture. More specifically, given any routing table containing N prefixes, each of length $\geq L$, we would like to calculate the size of the largest bucket generated by the best possible hash function that uses $k \leq L$ bits out of the first L bits for hashing ($L = 16$, from our assumptions).

Since a hash function uses k out of a candidate L bits, there are a total of $\binom{L}{k}$ possible hash functions. Given any prefix set of size N, we call the sum of the largest bucket sizes over all the $\binom{L}{k}$ hash functions to be *MaxSum* for the prefix set. Similarly, we define *ZeroSum* to be the sum of the sizes of the buckets where all the hashing bits have value zero (the *all-zero* buckets). We first show that among all the prefix sets of size N that maximize MaxSum, there is one where MaxSum is equal to ZeroSum. Next, we show that for *any* prefix set of size N, ZeroSum is at most $F(N, L, k)$; the function $F(N, L, k)$ is defined below. From this, we conclude that MaxSum can be at most $F(N, L, k)$, and thus there exists some hash function where the largest bin has size at most $\frac{F(N,L,k)}{\binom{L}{k}}$, which is the average size of the largest bucket over all hash functions. This gives as a worst-case bound on the power consumption.

Definitions. For any set of prefixes, we call the largest bucket created by the best hash function the *MaxBucket*. The worst case bound on the TCAM power consumption in the bit selection architecture for a table of size N is computed by considering the set of N prefixes for which MaxBucket is the largest. Since the hashing bits are selected from the first L bits of each prefix ($L = 16$ in our case), we represent a set of N prefixes as a set of L-bit weighted vectors, with a total weight of N. The weight $\mathrm{wt}(y)$ of an L-bit vector y is defined as the number of prefixes in the prefix set that have the first L bits the same as y. Let $\hat{w}$ be the maximum possible weight for a vector. Since we focus on 16-24 bit prefixes, $\hat{w} = 256$; this is the maximum number of 16-24 bit prefixes that share the first 16 bits, and are not completely covered by some other 16-24 bit prefix. Next, let supp(y) denote the support (number of non-zero bits) of the vector y. The function $F(N, L, k)$ is then defined as $F(N, L, k) = \hat{w} \sum_{a \in A} \binom{L-\mathrm{supp}(a)}{k}$, where $A \subset \{0, 1\}^L$ is a set consisting of the first $N/\hat{w}$ vectors in order of increasing support, each with weight $\hat{w}$.

Finally, we define H to be the set of all possible $\binom{L}{k}$ hash functions that use k bit positions to split the input set into 2^k buckets.

Upper bound on power consumption. Recall that the worst case power consumption of the TCAM is directly proportional to the size of the largest bucket. The following theorem states an upper bound on the size of the largest bucket for any input prefix set; the proof can be found in Appendix I.

Theorem III.1: For all $Y \subseteq \{0, 1\}^L$, $\sum_{y \in Y} \mathrm{wt}(y) = N$, there exists some hash function $h \in H$ that splits the set Y into buckets such that the size of the largest bucket is at most $\frac{F(N,L,k)}{\binom{L}{k}}$.

This bound is tight whenever $N/\hat{w} = \sum_{i \leq j} \binom{L}{i}$ for some value of $j \leq L$. In this case, for the set A, the largest bucket for every hash function has the same size, so the bucket sizes for the average and the best hash function coincide. However, for other values of $N/\hat{w}$, the upper bound given by the theorem is not tight. For example, adding one more prefix will raise the weight of the largest buckets on average, but may not affect the largest bucket for the best hash function.

We show the actual upper and lower bounds on the size of the largest bucket relative to the size of the prefix set for selected values of L, k, and N in Table I. For example, for any set of 1 million 16-24 bit prefixes (N=1M), there exists a 3-bit hash function (k=3) for which the biggest bucket is guaranteed not to contain more than 0.381M prefixes; this guarantees a power reduction of a factor by $1/.381 = 2.62$. In contrast, an ideal hash function would generate $2^k = 8$ equal-sized buckets, reducing the power consumption by a factor of 8.

N	100K	178177 (174K)	0.5M	644097 (0.6M)	1M	1762305 (1.68M)
	Upper	Lower	Upper	Lower	Upper	Lower
$k = 3$	0.566	0.542	0.444	0.434	0.381	0.345
$k = 6$	0.281	0.252	0.163	0.153	0.117	0.092

TABLE I. Lower and upper bounds on the size of the largest bucket for the Bit-Selection Scheme, $L = 16$; the sizes are relative to the number of prefixes N. Note that the lower bounds are given only for values of N where $N/\hat{w} = \sum_{i \leq j} \binom{L}{i}$ for some $j \leq L$, and are tight bounds (i.e., the same as upper bounds). For other values of N (such as 100K), we only provide the upper bounds, which are loose.

C. *The Bit Selection Heuristics*

In practice, we do not expect to find a real routing table that matches the worst-case input described in Section III-B. However, as explained earlier, the bound on the worst case input helps designers to determine the power budget. Given such a power budget, and a routing table, it is sufficient to ensure that the set of selected hashing bits produces a split that does not exceed the power budget. We call such a split a *satisfying split*. Note that it is possible that for the given routing table, a different partitioning (with lower power consumption) exists but we only care about keeping the power consumption below the power budget.

In this section, we describe three different heuristics for choosing the set of hashing bits. We then show how these heuristics can be combined to ensure that the power budget computed by Theorem III.1 can be maintained. Note that this methodology can be repeatedly applied to maintain the power budget when route updates occur.

Our first heuristic is the simplest (the *simple* heuristic) and requires no computation. This is based on the following observation. For almost all the routing table traces that we have analyzed, the rightmost k bits from the first 16 bits provide a satisfying split. However, this may not be true for tables that we have not examined or for tables of the future. Therefore, better schemes may be required if these hashing bits do not yield a satisfying split.

The second heuristic requires the most computation—it uses a brute force search to check all possible subsets of k bits from the first 16 bits and selects the first hashing set that satisfies the power budget. Obviously, this method is guaranteed to find a satisfying split. Since this method compares $\binom{16}{k}$ possible sets of k bits, its running time is maximum for $k = 8$.

Finally, the third heuristic is a greedy algorithm that falls between the brute force heuristic and the simple heuristic in terms of computation as well as accuracy. It may not find a satisfying split always, but has a higher chance of succeeding than the simple heuristic. To select k hashing bits, the greedy algorithm performs k iterations, selecting 1 hashing bit per iteration. Thus, the number of buckets (partitions of the routing table) doubles in each iteration. The goal in each iteration is to select a bit that minimizes the size of the biggest bucket produced by the 2-way split in that iteration (see Figure 2 for details).

We now outline a scheme that combines each of the three heuristics to minimize the running time of the bit-selection procedure. Let M be the lower bound on the worst-case size of the largest bucket (given by Theorem III.1), and T be the size of the entire TCAM. In addition, let P be the power consumption of the TCAM when all the entries are searched. Then the worst-case power budget is given by $P_b = (1+\alpha) \cdot \frac{M}{T} \cdot P$, where $0 < \alpha < 1$ provides a small additional margin for slack (say, 5%). It is possible to maintain a power budget of P_b using the following steps.

1) Split the routing prefixes using the last k of their first 16 bits. If this produces a satisfying split, stop.
2) Otherwise, apply the greedy heuristic to find a satisfying split using k hashing bits. If this produces a satisfying split, stop.
3) Otherwise, apply the brute force heuristic to find a satisfying split using k hashing bits.

We remind the reader that the algorithm described above must be applied whenever route updates change the prefix distribution in the routing table such that the size of the largest bucket exceeds M. For real tables, the expectation is that such recomputations will not be necessary very often. We explore the issue of recomputations in more detail in Section V.

D. Experimental results

In this subsection, we present experimental results of applying the bit selection heuristics described in Section III-C. We evaluated the heuristics with respect to two metrics—the running time of the heuristic, and the quality of the splits produced by the heuristics. For this purpose, we applied the heuristics to multiple real core routing tables,

```
$\mathcal{B} = \{\}$;
bins = $\{\mathcal{P}\}$;
**for** $i$ = 1 to $k$
    minmax = $\infty$;
    **foreach** bit $b \in \{1, \ldots, 16\} - \mathcal{B}$
        bins$_b$ = $\{s_{b=0}, s_{b=1} | s \in$ bins $\}$;
        max$_b$ = max (bins$_b$);
        **if** (minmax $>$ max$_b$) **then**
            min_bit = $b$;
            minmax = max$_b$;
        **endif**
    **endforeach**
    $\mathcal{B} = \mathcal{B} \cup$ min_bit;
    bins = bins$_{\text{min\_bit}}$
**endfor**
```

Fig. 2. Greedy algorithm for selecting k hashing bits for a satisfying split. $\mathcal{B}$ is the set of bits selected, and $\mathcal{P}$ is the set of all prefixes in the routing table. Here $s_{b=j}$ denotes the subset of prefixes in set s that have a value of j ($j = 0$ or 1) in bit position b.

site	location	date	table size
rrc04	Geneva	11/01/2001	109,600
oregon	Oregon	05/01/2002	121,883

TABLE II. The two core routing tables used to test the bit selection schemes.

and we present the results for 2 of those tables. Details of these routing tables are listed in Table II. The results of applying the algorithms to the other core routing tables were similar.

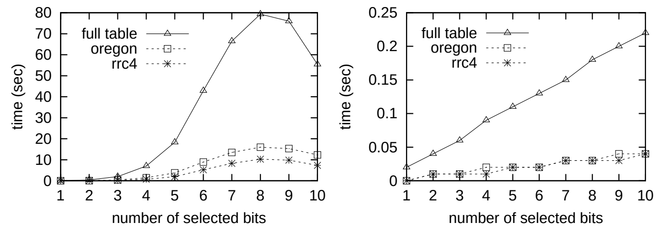
Running Times. The running times for the brute force and the greedy heuristics are shown in Figure 3.[1] All the experiments were run on a 800 MHz PC and required less than 1MB of memory. We first consider the running time of the brute force heuristic. For the real routing tables, there were less than 12,000 unique combinations of the first 16 bits for the 16-24 bit prefixes. The running time for the brute force algorithm was less than 16 seconds for selecting up to 10 hashing bits (Figure 3(a)).

To explore the worst case running times for 1M prefixes, we generated a synthetic table that has approximately 1 million prefixes with 2^{16} unique combinations of the first 16 bits. This table was constructed by randomly picking the (non-zero) number of prefixes that share each combination of the first 16 bits. In this case, the running time can go as high as 80 seconds for selecting 8 hashing bits.

Looking at the numbers for the greedy heuristic, we find that for real tables, it can run in as low as 0.05 seconds (up to 10 hashing bits) and takes about 0.22 seconds for the worst case synthetic input (Figure 3(b)). This is an order of magnitude faster than the brute force heuristic. However, if the routing updates do not require frequent reorganizations of the routing tables, the brute force method might also suffice.

Quality of Splits. We now explore the nature of the splits produced by each of the three heuristics. Let N denote the number of 16-24 bit prefixes in the table, and c_{max} denote the maximum bucket size. The ratio $\frac{N}{c_{max}}$ is a measure of the quality (evenness) of the split produced by the hashing bits. In particular, it is the factor of reduction in the portion of the TCAM that needs to be searched. Figure 4 shows a plot of $\frac{N}{c_{max}}$ versus the number of hashing bits k. From the figure,

[1]Note that the simple heuristic is a static selection process.



(a) brute force (b) greedy algorithm

Fig. 3. Running times of the brute force and greedy algorithms. The brute force algorithm performs an exhaustive search to find the best bits to select, while the greedy algorithm may find a suboptimal set of bits. "full" is the synthetic table, while rrc4 and oregon are real core routing tables.

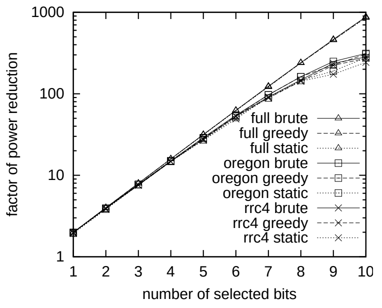


Fig. 4. Power reduction factor ($= \frac{\text{size of entire prefix table}}{\text{size of largest bucket}}$) plotted on a log scale, using the different algorithms. "brute" uses the brute force method, "greedy" uses the greedy algorithm, while "static" uses the last few consecutive bits out of the first 16 bits of a prefix.

we see that the ratio $\frac{N}{c_{max}}$ for the brute force and greedy schemes is nearly 53 at $k = 6$; for the static scheme this ratio is around 49, while this ratio for the best possible split (a completely even split) would be $2^k = 64$. The differences between the three bit selection heuristics widens as more hashing bits are used. Since the synthetic table was generated by selecting the number of prefixes for each combination of the first 16 bits uniformly at random, it is easier to find good hashing bits for it. Hence all the three bit selection schemes provide splits that are close to ideal for the synthetic table. In contrast, real tables are less uniform than the synthetic table yielding more uneven splits, and therefore, lower power reduction ratios.

Laying out buckets on TCAM blocks. We now consider the problem of laying out the buckets (corresponding to a satisfying split) on the TCAM blocks. First, the blocks containing the very long and very short prefixes are placed in the TCAM at the beginning and the end, respectively. This ensures that the longest prefix is selected in the event of multiple matches. We now focus on the buckets containing the 16-24 bit prefixes. Let the size of the largest bucket be c_{max}, and let the size of each TCAM block be s. Ideally, we would like at most $\lceil c_{max}/s \rceil$ blocks be to searched when any address is looked up. However, it is possible to show that for any TCAM with capacity N and block size s, there exists a possible split of N prefixes into buckets (of maximum size c_{max}) such that every possible layout scheme will have to lay out at least one bucket over $(\lceil c_{max}/s \rceil + 1)$ TCAM blocks.

Our scheme lays out the buckets sequentially in any order in the TCAM, ensuring that all the prefixes in one bucket are in contiguous locations. It is possible to show that for this scheme, each bucket of size c occupies no more than $\lceil c/s \rceil + 1$ TCAM blocks. Consequently, at most $\lceil c_{max}/s \rceil + 1$ TCAM blocks need to be searched during any lookup. Thus, our layout scheme is optimal in the sense that it matches the lower bound discussed in the previous paragraph.

The actual power savings ratio will be lower than the metric N/c_{max} plotted in Figure 4. This is because the bucket layout scheme may round up the number of searched blocks and the extra blocks containing the long and short prefixes need to be searched for every lookup. For example, consider the task of laying out a 512K-entry prefix table into a 512K-entry TCAM with 64 8K blocks. Suppose that the very short ($<$ 16-bit) and very long ($>$ 24-bit) prefixes fit into 2 blocks, while the biggest bucket contains 12K 16-24 bit prefixes. The metric N/c_{max} has the value $512K/12K = 42.67$. However, our layout scheme guarantees that the maximum number of blocks searched during a lookup would be $(\lceil 12K/8K \rceil + 1) + 2 = 5$, which reduces power consumption by a factor of $64/5 = 12.8$. For a TCAM with a maximum power rating of 15 Watts, this results in a power budget of under 1.2 Watts, which is in the same ballpark as the SRAM-based ASIC designs [3].

E. Discussion

The bit selection architecture provides a straightforward technique for reducing the power consumption of data TCAMs. In particular, the additional hardware required for bit extraction and hashing is a set of simple muxes and can be very cost effective. However, the technique has some drawbacks. First, the worst-case power consumption guaranteed by this method is fairly high. In practice (i.e., for real tables), we saw that our heuristics provide significantly lower power consumption. For example, for a table with N=1M prefixes, the worst-case analysis guarantees a power reduction ratio $N/c_{max} = 2.62$ using 3 hashing bits (from Table I), while our experimental results indicate power reduction ratios over 7.5 (from Figure 4). However, for a hardware designer who allocates a power budget, the worst-case power requirement is required to provide a guaranteed-not-to-exceed power budget. Thus, for the bit selection architecture, the designer would *be forced* to design for a much higher worst-case power consumption than will ever be seen in practice.

Second, the method of bit-selection described here assumes that the bulk of the prefixes lie in the 16-24 bit range[2]. This assumption may not hold in the future. In particular, the number of long ($>$ 24-bit) prefixes may increase rapidly in the future [2]. In the next section, we present tree-based algorithms that do not make any assumptions regarding the distribution of prefixes to be split and provide tighter bounds on worst-case power consumption at the cost of some additional hardware.

IV. TRIE-BASED TABLE PARTITIONING

We now present two schemes that use a routing tree (*trie*) data structure for partitioning the routing table into TCAM buckets. Both these schemes eliminate the two main drawbacks of the bit-selection architecture that we discussed in Section III-E. In other words, the trie-based schemes do not assume that the bulk of the prefixes lie in the 16-24 bit range, and provide bounds on the worst-case power budget that are matched in practice. Since it is no longer assumed that most prefixes lie in the 16-24 bit range, the trie-based schemes are able to

[2]For example, the upper bounds on power consumption from Table I would increase if these assumptions are violated.

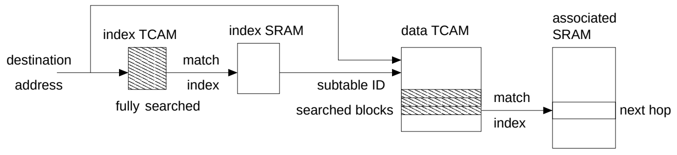


Fig. 5. Forwarding engine architecture for the trie-based power reduction schemes.

partition the entire routing table, instead of concentrating only on the 16-24 bit prefixes.

As mentioned earlier, the main difference between the bit-selection architecture and the trie-based architecture is that the latter uses a prefix trie in the first stage lookup process instead of hashing on a set of input bits. This requires some additional hardware, as shown in Figure 5. Each input is first matched with respect to an initial, small-sized TCAM (containing the trie) that indexes into an associated SRAM. We call these the *index TCAM*, and the *index SRAM*, respectively. The index SRAM contains the ID of the TCAM bucket (obtained as a result of the partitioning) that should be searched in the second stage lookup. Obviously, this lookup process requires the entire index TCAM to be searched every time. However, we shall show that the index TCAM is typically very small in size compared to the data TCAM, and does not contribute significantly to the power budget. As with the bit-selection architecture, a large routing table can be laid out across multiple data TCAMs connected in parallel. Each address lookup then involves searching exactly one bucket in one of these data TCAMs. For simplicity, we shall assume that there is only one data TCAM—our results apply equally well when there are multiple data TCAMs.

The forwarding architecture described above raises two important issues. First, we must show how the trie in the index TCAM is constructed, and second, we must calculate the upper bounds on the size of the index TCAM (since this is a component of the power budget). We address both these issues in the next few subsections where we present the two trie-based partitioning schemes. Finally, we note that the second partitioning scheme provides a better (or lower) bound on the worst case bucket sizes of the data TCAM at the expense of a larger index TCAM.

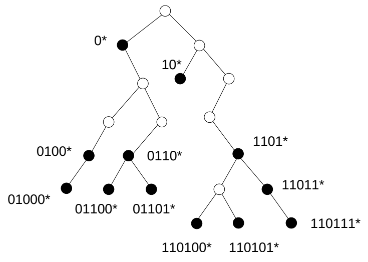
A. Overview of Routing Tries

The partitioning schemes that we present here both work in two steps. First a binary routing trie (often called a *1-bit trie* [10]) is constructed using the routing table. In the second step, subtrees or collections of subtrees of the 1-bit trie are successively carved out and mapped to individual TCAM buckets. We call this the *partitioning step*. The two partitioning schemes essentially differ in their partitioning step.

Before we present the actual schemes, we first present an overview of 1-bit tries and some of their important properties that we utilize. A 1-bit trie is used for performing longest prefix matches. It consists of a collection of nodes, where a routing prefix of length n is stored at a node in level n of the trie. When presented with an input, the lookup process starts from the root, scans the input from left to right and descends the left (right) branch when the next bit in the input is 0 (1) until a leaf node is reached. This traces a path from the root node to the longest prefix that matches the input. For any node v, the prefix denoted by the path from the root to v is called the *prefix of* v, and the number of routing table prefixes in the subtree rooted at v is called the *count of* v.

0$*$
0100$*$
01000$*$
0110$*$
01100$*$
01101$*$
10$*$
1101$*$
110100$*$
110101$*$
11011$*$
110111$*$

(a)



(b)

Fig. 6. (a) An example routing table, and (b) the corresponding 1-bit trie built from it. The prefixes of only the black nodes are in the routing table. A "$*$" in the prefixes denotes the start of don't-care bits.

Every node in the 1-bit trie must have a non-zero count, i.e., a node appears in the trie only if the subtree rooted at that node contains at least one prefix. Therefore, the prefix of a leaf node must be in the routing table. In contrast, the prefix of an intermediate node may not be in the routing table. For any node u, the prefix of the lowest common ancestor of u (including u itself) that is in the routing table is called the *covering prefix* of u. The covering prefix of a node is nil if there are no nodes in the path to the root whose prefix is in the routing table. Figure 6 presents an example 1-bit trie built from a routing table.

B. Splitting into subtrees

We now describe the first of the two trie-based partitioning algorithms, called ***subtree-split*** (see Figure 7). This algorithm takes as input a parameter b that denotes the maximum size of a TCAM bucket (in terms of number of prefixes). The output is a set of $K \in [\lceil \frac{N}{b} \rceil, \lceil \frac{2N}{b} \rceil]$ TCAM buckets, each with a size in the range $[\lceil b/2 \rceil, b]$, and an index TCAM of size K. During the partitioning step, the entire trie is traversed in post order looking for a *carving node*. A carving node is a node v whose count is at least $\lceil b/2 \rceil$ and whose parent exists and has a count greater than b. Every time a carving node v is encountered, the entire subtree rooted at v is carved out and placed in a separate TCAM bucket. Next, the prefix of v is placed in the index TCAM, and the covering prefix of v is added to the TCAM bucket (we explain why in the next paragraph). Finally, the counts of all the ancestors of v are decreased by the count of v. In other words, once the subtree rooted at v is carved out, the state of the rest of the tree is updated to reflect that. When there are no more carving nodes left in the trie, the remaining prefixes (if any) are put in a new TCAM bucket with an index entry of $*$ in the index TCAM. Note that the size of this last TCAM bucket is in the range $[1, b]$.

Figure 8 shows how subtrees are carved out of the 1-bit trie from Figure 6. Note that the index (root) for a carved subtree need not hold a prefix from the routing table. Hence the index TCAM may include prefixes not in the original routing table. They simply serve as pointers to the buckets in the data TCAM that contains the corresponding routing table prefixes. Therefore an input address that matches an entry in the index TCAM may have no matching prefix in the corresponding subtree. The addition of the covering prefix to a bucket ensures that a correct result is returned in this case. For example, for the partitioning in Figure 8, the input address 01011111 matches 010$*$ in the index

```
subtree-split($b$):
    while (there is a next node in post order)
        $p$ = next node in post order;
        if (count($p$) $\geq \lceil b/2 \rceil$ and
           (count(parent($p$)) $> b$) then
            carve out subtree rooted at $p$
            put subtree in new TCAM bucket $bu$
            put prefi x($p$) in index TCAM
            $bu = bu \cup \{cp(p)\}$
            foreach node $u$ along path from root to $p$
                count($u$) = count($u$) $-$ count($p$)
                if (count($u$) == 0) then remove $u$ endif
            endforeach
        endif
    endwhile
```

Fig. 7. Algorithm subtree-split for carving the 1-bit trie into buckets of size in the range $[\lceil b/2 \rceil, b]$. Here count(p) is the number of prefixes remaining under node p, prefix(p) is the prefix of node p, parent(p) is the parent node of p in the 1-bit trie, and cp(p) is the covering prefix of node p.

TCAM, but has no matching prefix in the corresponding subtree. The covering prefix $0*$ is the correct longest matching prefix for this input.

Since we perform a post order traversal of the trie, the subtree indices must be added to the index TCAM *in the order* that the corresponding subtrees were carved out. In other words, the first subtree index must have the highest priority (lowest address) in the index TCAM, while the last subtree index must have the lowest priority. Finally, each bucket can be laid out in the data TCAM as described in Section III-C.

The following properties can be proved for algorithm subtree-split when applied with parameter b to a table with N prefixes; *the proofs have been omitted due to lack of space.*

Theorem IV.1: The size of each bucket created lies in the range $[\lceil b/2 \rceil, b]$, *except for the last bucket, whose size is in the range* $[1, b]$. *In addition, at most one covering prefix is added to each bucket.*

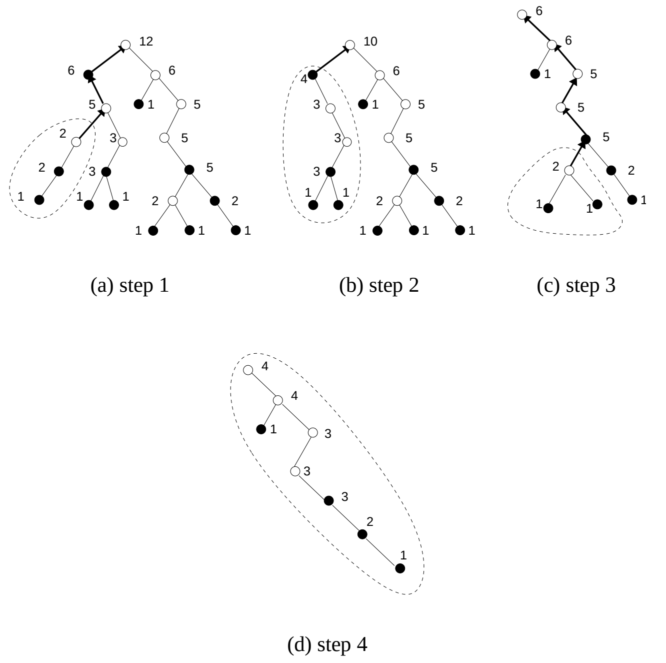
Theorem IV.2: The total number of buckets created is in the range $[\lceil \frac{N}{b} \rceil, \lceil \frac{2N}{b} \rceil]$. *Each bucket results in one entry in the index TCAM and one entry in the index SRAM.*

Theorem IV.3: The index and data TCAMs populated according to the subtree-split algorithm always return the longest matching prefix for each input address.

Finally, to split N prefixes into K buckets, subtree-split is run with parameter $b = \lceil 2N/K \rceil$. Since the maximum bucket size (including the covering prefix) is $b + 1$, we have:

Theorem IV.4: Using subtree-split in a TCAM with K buckets, during any lookup at most $K + \lceil 2N/K \rceil + 1$ prefixes are searched from the index and the data TCAMs.

Complexity. The post-order traversal for the 1-bit trie implies that each node in the trie is encountered at most once during the traversal. For a routing table with N prefixes, the number of nodes in the corresponding 1-bit trie is $O(N)$. Therefore, the complexity for this part of the algorithm is $O(N)$. Every time a subtree is carved out, we need to traverse the 1-bit trie all the way to the root. The number of subtrees carved out is $O(N/b)$ (from Theorem IV.2). If W is the maximum prefix length, (hence, maximum trie depth), this gives us a complexity of $O(NW/b)$. Finally, the total work for laying out the routing table in the TCAM buckets is $O(N)$ (each routing table prefix is looked at once). Thus the total complexity of the algorithm is $O(N + NW/b)$.


(a) step 1 (b) step 2 (c) step 3

(d) step 4

index	bucket prefi xes	bucket size	covering prefi x
$010*$	$0100*$, $01000*$	2	$0*$
$0*$	$0*$, $0110*$, $01100*$, $01101*$	4	$0*$
$11010*$	$110100*$, $110101*$	2	$1101*$
$*$	$10*$, $1101*$, $11011*$, $110111*$	4	—

(e) fi nal partitioning

Fig. 8. (a), (b), (c), (d) : four iterations of the subtree-split algorithm (with parameter b set to 4) applied to the 1-bit trie from Figure 6. The number at each node u denotes the current value of count(u). The arrows show the path along which count (u) is updated in each iteration, while the dashed outline denotes the subtree that is carved. The table in (e) shows the five resulting buckets. Bucket sizes vary between $b/2$ and b prefixes. The covering prefix of each bucket, if not already in the bucket, is finally added to it.

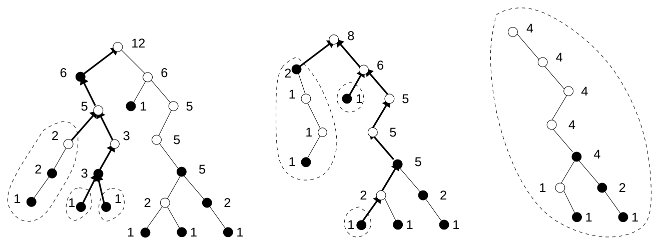
C. Post-order splitting

The drawback with algorithm subtree-split is that the smallest and largest bucket sizes vary by as much as a factor of 2. In this section, we introduce another trie splitting algorithm called ***postorder-split*** that remedies this. Once again, let N be the total size of a routing table, and b be the desired size of a TCAM bucket. The algorithm postorder-split partitions the routing table into buckets that each contain exactly b prefixes (except possibly the last bucket). Such an even partitioning comes at the extra cost of a larger number of entries in the index TCAM.

The main steps in the postorder-split algorithm are similar to that in the subtree-split algorithm. It first constructs a 1-bit trie from the routing table and then traverses the trie in post-order, carving out subtrees to put in TCAM buckets. However, it is possible that the entire trie does not contain $\lceil \frac{N}{b} \rceil$ subtrees with exactly b prefixes each. Since each resulting TCAM bucket must be of size b, a bucket here is constructed from a collection of subtrees which together contain exactly b prefixes, rather than a single subtree (as in the case of algorithm subtree-split).

```
postorder-split($b$):
    $i$ = 0;
    while (there is a next node in post order)
        $p$ = next node in post order
        carve-exact ($p, b, i$)
        $i = i + 1$
    endwhile
carve-exact ($p, s, i$):
        if (count($p$) == $s$) or
           (count($p$) $< s$ and
           count (parent($p$)) $> s$) then
             carve out subtree rooted at $p$
             put subtree in TCAM bucket $bu_i$
             put prefix($p$) in index TCAM ($index_i$)
             $bu_i = bu_i \cup \{cp(p)\}$
             foreach node $u$ along path from root to $p$
                 count($u$) = count($u$) $-$ count($p$)
                 if (count($u$) == 0) then remove $u$ endif
             endforeach
             if (count($p$) $< s$) then
                 $x$ = count($p$)
                 $q$ = next node in post order
                 carve-exact($q$, $s - x, i$)
             endif
        endif
```

Fig. 9. Algorithm postorder-split for carving the 1-bit trie into buckets of size b. Here, count(p) is the number of prefixes remaining under node p, prefix(p) is the prefix of node p, parent(p) is the parent node of p in the 1-bit trie, and cp(p) is the covering prefix of node p. Here $index_i$ is the set of entries in the index TCAM that point to the bucket bu_i in the data TCAM.


(a) step 1 (b) step 2 (c) step 3

i	index$_i$	bucket prefixes	$size$	cover
1	010$*$, 01100$*$, 01101$*$	0100$*$, 01000$*$, 01100$*$, 01101$*$	4	0$*$, 01100$*$, 01101$*$
2	0$*$, 10$*$, 110100$*$	0$*$, 0110$*$, 10$*$, 110100$*$	4	0$*$, 10$*$ 110100$*$
3	1$*$	110101$*$, 1101$*$, 11011$*$, 110111$*$	4	—

(d) final partitioning

Fig. 10. (a), (b), (c): three iterations of the postorder-split algorithm (with parameter b set to 4) applied to the 1-bit trie from Figure 6. The number at each node u denotes the current value of count(u). The arrows show the path traced to the root in each iteration for decrementing count. The dashed outlines denote the set of subtrees carved out. (d) the three resulting buckets. Each bucket has $b = 4$ prefixes. The covering prefixes of each bucket that are not in the bucket are finally added to the bucket in the data TCAM.

Consequently, the corresponding entry in the index TCAM has multiple indices that point to the same TCAM bucket in the data TCAM. Each such index is the root of one of the subtrees that constitutes the TCAM bucket.

The algorithm postorder-split is shown in Figure 9. The outer loop (procedure postorder-split) traverses the 1-bit trie in post-order and successively carves out subtree collections that together contain exactly b prefixes. The inner loop (procedure carve-exact) performs the actual carving—if a node v is encountered such that the count of v is b, a new TCAM bucket is created, the prefix of v is put in the index TCAM and the covering prefix of v is put in the TCAM bucket. However, if the count of v is x such that $x < b$ and the count of v's parent is $> b$, then a recursive carving procedure is performed. Let the node next to v in post-order traversal be u. Then, the subtree rooted at u is traversed in post-order, and the algorithm attempts to carve out a subtree of size $b - x$ from it. In addition, the x entries are put into the current TCAM bucket (a new one is created if necessary), and the prefix of v is added to the index TCAM and made to point to the current TCAM bucket. The covering prefix of v is also added to the current TCAM bucket. Finally, when no more subtrees can be carved out in this fashion, the remaining prefixes, if any (they must be less than b in number), are put in a new TCAM bucket and a $*$ entry in the index TCAM points to the last bucket. Figure 10 shows a sample execution of the algorithm.

Note that this algorithm may add more than one index (and covering) prefix per TCAM bucket. The number of prefixes added to the index TCAM for any given TCAM bucket is equal to the number of times the carve-exact procedure is called recursively to create that bucket. It is possible to show that each time carve-exact is called for this bucket, we descend one level down in the 1-bit trie (except, possibly, for the last invocation of carve-exact). Therefore, the maximum number of times we can call the carve-exact procedure is $W + 1$, where W is the maximum prefix length in the routing table. In other words, the algorithm postorder-split adds at most $W + 1$ entries to the index TCAM and W covering prefixes to the bucket in the data TCAM.

The following properties can be proved about algorithm postorder-split when applied with parameter b to a table with N prefixes of maximum length W bits.

Theorem IV.5: The size of each bucket created by the postorder-split algorithm is b, except for the last bucket, whose size is in the range $[1, b]$. At most W covering prefixes are added to each bucket.

Theorem IV.6: The number of buckets created by postorder-split is exactly $\lceil \frac{N}{b} \rceil$. Each bucket contributes at most $W + 1$ entries to both the index TCAM and the index SRAM.

Theorem IV.7: The index and data TCAMs populated according to the postorder-split algorithm always return the longest matching prefix for each input address.

To split N prefixes of maximum length W into K buckets, postorder-split is run with parameter $b = \lceil N/K \rceil$. Therefore, the following bound holds.

Theorem IV.8: Using postorder-split in a TCAM with K buckets, during any lookup at most $(W + 1)K + \lceil N/K \rceil + W$ prefixes are searched from the index and the data TCAMs.

Complexity. The complexity analysis of the postorder-split algorithm is similar to that of the subtree-split algorithm, and it is possible to show that the total running time of the algorithm is $O(N + NW/b)$, where W is the maximum prefix length.

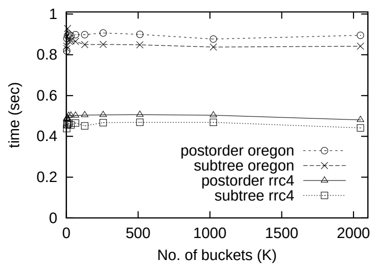
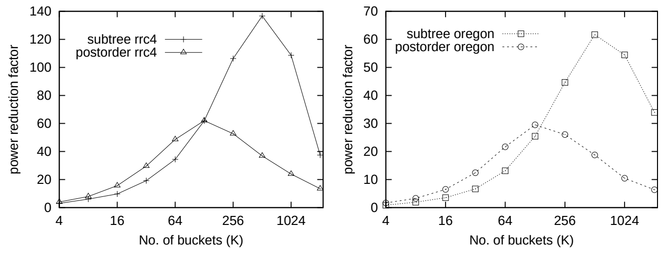


Fig. 11. Running time to build and split a 1-bit trie using algorithms subtree-split and postorder-split on the rrc and oregon routing tables.



(a) rrc (b) oregon

Fig. 12. Reduction in maximum number of routing table entries searched when algorithms subtree-split and postorder-split are applied to (a) the rrc and (b) the oregon routing tables.

D. Experimental results

In this section, we present some results of applying algorithms subtree-split and postorder-split to the routing tables listed in Table II. Instead of the input parameter b which limits the size of each resulting bucket, we implemented the algorithms to take as input the total number of buckets K.

The time taken to partition the routing table into K buckets is shown in Figure 11. This includes the time required to build a 1-bit trie from the routing table entries. Both algorithms complete within one second for the routing tables from Table II. Approximately one third of the running time is spent in building the 1-bit trie in each case. Further, the running time does not increase significantly for larger numbers of buckets. Algorithm subtree-split is marginally faster than postorder-split, since it finds only one subtree in the 1-bit trie to carve out in each iteration (to create a bucket).

The reduction in power consumption using algorithms subtree-split and postorder-split is shown in Figure 12. The figure plots the ratio of the total number of routing table prefixes to the maximum number of prefixes searched during the lookup process. This maximum is computed as the sum of the total number of entries in the index TCAM (since it is always searched in full) and the number of entries in the largest bucket in the data TCAM (since only one such bucket is searched for every lookup). When the total number of buckets K in the TCAM is small (this is often a limitation on commercially available TCAMs), postorder-split performs better since it generates a more even split into buckets. However, as K grows beyond 64, the size of the index TCAM begins to dominate in the case postorder-split, which may add up to W=32 entries in the index TCAM for each bucket. In contrast, although subtree-split provides a less even split, it only adds one entry to the index TCAM for each bucket, and the index TCAM remains small as K increases. It is also interesting to note that for both the algorithms, this trade-off implies that there is an optimal number of buckets for which the worst-case power reduction factor is highest: $\sqrt{2N}$ buckets for subtree-split and $\sqrt{N/W}$ for postorder-split.

All three methods described in this paper are successful in reducing power consumption in practice. For example, for the rrc routing table, if the data TCAM is limited to have 8 buckets, bit selection using the exhaustive and greedy algorithms reduces the power consumption by factors of 7.58 and 7.55, respectively. The subtree-split and postorder-split algorithms result in power reduction factors of 6.09 and 7.95, respectively (including the power consumed by the index TCAM). Thus, in practice, a data TCAM that consumes a maximum of 15 watts needs less than 2.5 watts using any of these algorithms. Similarly, if the data TCAM supports 64 buckets, the power consumption can be reduced to less than 0.5 watts.

However, if a *worst case* power budget is required while designing the forwarding engine, the trie-based algorithms are better than the bit selection schemes. As an example, consider a data TCAM that can support 8 buckets, and an IPv4 routing table with 1 million entries (here $W = 32$). Algorithm postorder-split can guarantee a split of the routing table into buckets, such that each bucket has exactly $10^6/8 = 125,000$ entries. Adding in up to 32 covering prefixes for the bucket, and an index TCAM with up to $8 \times 32 = 256$ entries, a maximum of only 125,288 prefixes need to be searched during each lookup, resulting in power reduction by a factor of $10^6/125,288 = 7.98$; this reduced power budget is independent of the distribution of prefixes in the routing table. In contrast, bit selection can guarantee a power reduction factor of only $1/.381 = 2.62$, assuming most prefixes are 16-24 bits long. Thus, with a 15 watt 1M-entry data TCAM, postorder-split would result in a power budget of under 2 watts, while bit-selection can only guarantee a power budget of around 5.7 watts.

V. Route table updates

In this section, we briefly explore the performance of the bit selection and the trie-based architectures in the face of routing table updates (route additions and withdrawals). Adding routes (prefixes) may cause a bucket in the data TCAM to overflow, requiring a repartitioning of the prefixes into buckets and rewriting the entire table in the data TCAM. Therefore, the number of repartitions should be minimized. We applied real-life updates traces (each with a few million route updates) collected at the same times and sites as the routing tables in Table II. The heuristics to avoid frequent repartitions for each architecture are described below.

For the bit selection architecture, we started by applying the brute force heuristic on the initial table, and noted the size c_{max} of the largest bucket. Using a fixed threshold t, we recomputed the hashing bits every time the largest bucket size exceeded $c_{thresh} = (1 + t) \times c_{max}$. The hashing bits were recomputed using the static (last few bits) heuristic, followed by the greedy heuristic if the threshold was not met. If the greedy heuristic also failed to bring the maximum bucket size under c_{thresh} we applied the brute force heuristic, and updated the values of c_{max} and c_{thresh}. The number of times each heuristic was applied is listed in Table III; the "static" column represents the total number of recomputations required over the course of the updates.

A similar threshold-based strategy was applied to the trie-based architecture; as before, we assume buckets need to be recomputed each

		rrc4			oregon		
#updates		3,412,540			3,614,740		
init size		107,195			119,226		
final size		103,873			113,436		
buckets	thresh	static	greedy	brute	static	greedy	brute
8	1	2	1	1	15	14	9
8	5	0	0	0	3	2	1
8	10	0	0	0	1	0	0
64	1	12	12	12	13	12	11
64	5	6	5	2	7	5	3
64	10	2	1	0	3	2	1

TABLE III. Number of times each heuristic is reapplied during the course of route table updates. "buckets' are the number of buckets created: 3 (or 6) hashing bits create 8 (or 64) buckets. "thresh" is the threshold t (in percent) by which the size of the maximum bucket is allowed to grow before the bits are recomputed. "init" and "final" denote the number of 16-24 bit prefixes before and after the updates are applied.

time any bucket overflows. The results are shown in Table IV. The update traces contain occasional floods of up to a few thousand route additions in a single second, where the new routes are very close to each other in the routing trie (they are often subsequently withdrawn). Although rare, these floods often cause a single bucket in the trie-based architecture to repeatedly overflow, since prefixes close together in the routing trie are placed in the same bucket in the trie-based partitioning algorithms. In contrast, the bit selection scheme spreads out nearby prefixes across multiple buckets (since it selects a subset of prefix bits for indexing to TCAM buckets) and therefore requires far fewer repartitions. For the subtree-split and postorder-split algorithms we used bucket sizes of $\lceil 2N/K \rceil$ and $\lceil N/K \rceil$, respectively, for N prefixes and K buckets; therefore subtree-split required significantly fewer recomputations than postorder-split.

		rrc4			oregon		
buckets	thresh	sub-tree	post-order	post-opt	sub-tree	post-order	post-opt
8	1	0	58	3 (.07)	3	74	2 (.07)
8	5	0	17	2 (.05)	1	14	1 (.04)
8	10	0	0	0 (.01)	0	6	1 (.03)
64	1	41	1957	236 (.48)	14	1042	84 (.56)
64	5	24	1019	152 (.44)	12	533	40 (.45)
64	10	20	649	109 (.37)	11	172	11 (.32)

TABLE IV. Number of times the buckets are recomputed when the update traces from Table III are applied to the trie-based architectures. "post-opt" uses the postorder split algorithm but is optimized for updates. The numbers in parentheses () denote the average additional TCAM writes per update (in addition to the minimum one data TCAM write) for transferring prefixes across neighboring buckets in the optimized scheme.

The postorder-split algorithm partitions prefixes after arranging them in order of postorder traversal. Hence transferring prefixes between neighboring buckets is straightforward. Such a local transfer requires a small number of writes to the data and index TCAMs. Therefore, one way to mitigate the problem of frequent repartitions using the postorder-split algorithm is to simply transfer some prefixes from the overflowing bucket to one of its neighbors, and recompute the entire partitioning only when both the neighboring buckets become full. On a recomputation, we also set the size of the overflowed bucket to zero, so that it can absorb a larger number of subsequent prefix additions.[3]

[3]We assume that future additions will hit the same TCAM bucket which is mostly the case, at least in the short term.

This solution, shown as "post-opt" in Table IV, reduces the number of recomputations. Transfers between neighboring buckets in the event of bucket overflows result in some additional TCAM writes; the numbers in parenthesis in Table IV denote the average TCAM writes per route update due to such transfers. For example, with 8 buckets and a 1% recomputation threshold in the optimized postorder-split scheme, each route update results in 1.07 TCAM writes instead of 1 TCAM write required by the other schemes.

Floods of a very large number of route updates are probably due to reboots of neighboring routers or BGP misconfigurations [6], but they do appear to occur every few days in real-life traces. The bit selection architecture has a natural advantage compared to trie-based schemes in the face of such events. But overall, both architectures in practice require a limited number of recomputations in the face of millions of route updates.

VI. Related work[4]

One of the earliest works to introduce ternary CAMs was by Wade and Sodini [12]—later, the authors proposed using TCAMs in a hardware search engine called a Database Accelerator [13]. The use of TCAMs for routing table lookups was first proposed by McAuley and Francis [7]; they also described the problem of updating TCAM-based routing tables that are sorted with respect to prefix lengths. Two solutions for this problem of updating TCAM-based routing tables have been proposed more recently [9]. Kobayashi et al. suggested associating each TCAM entry with a priority [4], and additional hardware was used to output the entry with the highest priority in case of multiple matches. This eliminated the sorting requirement but added extra latency to lookup operations.

The problem of high cost and power consumption in TCAMs was studied by Liu [5]. The author used a combination of pruning techniques (to eliminate redundant prefixes) and logic minimization algorithms to reduce the size of TCAM-based routing tables—in turn, this reduces the cost and power consumption of these devices. Finally a method was proposed for multi-field packet classification using TCAMs, where a pre-processing step is used to reduce the size of the TCAM [11].

VII. Summary and Discussion

We presented two alternatives for building low-power TCAM-based forwarding engines: the bit-selection architecture and the trie-based architecture. Both architectures rely on partitioning the route lookup table into small portions, so that only one portion needs to be searched for each lookup. For each architecture, we provide schemes for finding a good partitioning, based on the contents of the routing table. These partitioning schemes are fast and effective: they generate good (close to equal-sized) partitions in practice for real-life routing tables, and require infrequent repartitioning in the face of real-life updates. However, the trie-based schemes provide better worst-case upper bounds on power consumption, independent of the table contents. This can be an important advantage for hardware designers who need to fix a maximum power budget for a given table size, independent of the table contents.

[4]There has been a significant amount of research on lookup algorithms for software or ASIC-based forwarding engines; we omit those references here due to lack of space.

Alternate ways to design the first stage of the two-stage lookup architecture are possible, but beyond the scope of this paper. For example, a trie-based search in SRAM could be used to select a bucket in the data TCAM. The two-stage lookup architectures can also be generalized to multiple stages. For example, in the trie-based partitioning schemes, a three-stage lookup can be implemented by using two index TCAMs: the first index TCAM can select a bucket to search in the second index TCAM. However, the bulk of the power savings are achieved with a two-stage architecture, and additional stages provide only incremental savings beyond that. Finally, the complexity of the trie-based lookup architecture can be reduced by using a bucket in the data TCAM to act as the index TCAM; however, this would reduce the throughput of the lookup process by half, since the data TCAM must be accessed twice for each lookup.

ACKNOWLEDGMENTS

We would like to thank Jayesh Patel, Cliff Young, and the anonymous referees for valuable feedback on this work.

REFERENCES

[1] Anthony Gallo. Meeting Traffic Demands with Next-Generation Internet Infrastructure. *Lightwave*, 18(5):118–123, May 2001. Available at http://www.siliconaccess.com/news/Lightwave_may_01.html.

[2] G. Huston. Analyzing the Internet's BGP Routing Table. *The Internet Protocol Journal*, 4, 2001.

[3] IDT. `http://www.idt.com/products/`.

[4] M. Kobayashi, T. Murase, and A. Kuriyama. A Longest Prefix Match Search Engine for Multi-Gigabit IP Processing. In *Proceedings of the International Conference on Communications (ICC 2000)*, pages 1360–1364, New Orleans, LA, 2000.

[5] H. Liu. Routing Table Compaction in Ternary CAM. *IEEE Micro*, 22(1):58–64, January–February 2002.

[6] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *Proceedings of SIGCOMM '02*, Pittsburgh, PA, August 2002.

[7] A. J. McAuley and P. Francis. Fast Routing Table Lookup Using CAMs. In *Proceedings of Infocom '93*, pages 1382–1391, San Francisco, CA, March 1993.

[8] Netlogic microsystems. `http://www.netlogicmicro.com`.

[9] D. Shah and P. Gupta. Fast Updating Algorithms for TCAMs. *IEEE Micro*, 21(1):36–47, January–February 2001.

[10] V. Srinivasan and G. Varghese. Fast Address Lookups Using Controlled Prefix Expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, February 1999.

[11] J. van Lunteren and A. P. J. Engbersen. Multi-Field Packet Classification Using Ternary CAM. *Electronics Letters*, 38(1):21–23, 2002.

[12] J. P. Wade and C. G. Sodini. Dynamic Cross-Coupled Bitline Content Addressable Memory Cell for High Density Arrays. *IEEE Journal of Solid-State Circuits*, 22(2):119–121, February 1987.

[13] J. P. Wade and C. G. Sodini. A Ternary Content Addressable Search Engine. *IEEE Journal of Solid-State Circuits*, 24(4):1003–1013, August 1989.

APPENDIX I
ANALYSIS OF WORST CASE POWER CONSUMPTION

To develop the proof for Theorem III.1, we first introduce some additional notation. For a set Y of vectors, we define $\mathrm{wt}(Y) = \sum_{y \in Y} \mathrm{wt}(y)$. We now define a *bucket vector* B to be an assignment of a bucket (from $\{0,1\}^k$) for each hash function $h \in H$. The entry $B_h \in \{0,1\}^k$ denotes the bucket assigned to h; the bucket vector B has $|H| = \binom{L}{k}$ such entries. Given a bucket vector B and a set Y of weighted vectors (prefixes), we define $S(Y, B, h)$ to be the total weight of the elements in Y that are mapped into bucket B_h by the hash function h. Let $T(Y,B) = \sum_{h \in H} S(Y,B,h)$; then $\hat{T}(Y) = \max_B T(Y,B)$ is MaxSum, the sum of the sizes of the largest buckets over all hash functions $h \in H$.

Lemma I.1: There exists $Y \subseteq \{0,1\}^L$ such that $\mathrm{wt}(Y) = N$, Y maximizes $\hat{T}(Y)$ and the largest bucket is the all-zero bucket for every $h \in H$.

Proof: We first define an operation $G(Y,B,i) = (Y', B')$ such that Y' is obtained from Y by setting the ith bit position to 0 for all $y \in Y$, with one exception: if there are two inputs (called a *pair*) in Y which are identical on all positions except the ith, we set the ith bit position in the input with greater weight to 0, while the ith bit position in the other input is set to 1. Similarly, B' is such that for any $h \in H$, B'_h is obtained from B_h by setting to 0 the location (in B_h) corresponding to the ith bit position in the input. Note that if h does not use the ith bit in the input as a hashing bit, $B_h = B'_h$.

The operation G has the following properties:

1) $\mathrm{wt}(Y) = \mathrm{wt}(Y')$.
2) $\forall h \in H$, B'_h is 0 on the position corresponding to the ith input bit.
3) $\forall h \in H, S(Y', B', h) \geq S(Y, B, h)$.

The first two are immediate from the definition. For the third, observe that any $y \in Y$ which is sent to B_h is sent to B'_h afterwards unless it belongs to a pair. If one of a paired input is mapped to B_h by h, then there must be one that is mapped to B'_h by h, and the paired input with the larger weight gets a 0 in that position.

Let Y^0 be some input set of vectors maximizing the sum of the sizes of the largest buckets subject to the constraint that $\mathrm{wt}(Y^0) = N$, and let B^0 be the corresponding vector of largest buckets for Y^0. We now construct sequences $Y^0, \ldots, Y^L$ and $B^0, \ldots, B^L$ such that for $\forall i \in [1, L]$, $(Y^i, B^i) = G(Y^{i-1}, B^{i-1}, i)$. We then have $\mathrm{wt}(Y^L) = N$ (by the first property), $\forall h \in H, B^L_h$ is the all-zeroes bucket (by the second property), and $S(Y^L, B^L, h) \geq S(Y^0, B^0, h)$ (by repeated application of the third property). Thus $T(Y^L, B^L) \geq T(Y^0, B^0)$.

Let $T^* = T(Y^0, B^0)$ be the maximum value obtained by T over all sets with N vectors. We now know that

$$T^* = T(Y^0, B^0) \leq T(Y^L, B^L) \leq \hat{T}(Y^L) \leq T^*$$

where the last inequality follows from the maximality of T^*. Thus the input Y^L makes T as large as possible. At the same time, it makes the all-zero buckets the largest bucket for every hash function; otherwise, replacing the all-zero bucket with another bucket in B^L would increase the value of $T(Y^L, B^L)$. ■

Lemma I.2: For all $Y \subseteq \{0,1\}^L, \mathrm{wt}(Y) = N$, $\hat{T}(Y)$ is at most $F(N, L, k) = \hat{w} \sum_{a \in A} \binom{L - \mathrm{supp}(a)}{k}$, where $A \subset \{0,1\}^L$ is a set consisting of the first $N/\hat{w}$ vectors in order of increasing support, each with weight $\hat{w}$.

Proof: From lemma I.1, $\max_Y \hat{T}(Y) = \max_Y T(Y, B_z)$, where B_z is the bucket vector that assigns the all-zeroes bucket to every $h \in H$. Thus, it suffices to show an upper bound on $\max_Y T(Y, B_z)$.

Each vector $y \in Y$ contributes $\mathrm{wt}(y)$ to the all-zero bucket for every hash function that uses the hashing bits on which y is zero. Thus, each input y contributes precisely $\mathrm{wt}(y) \cdot \binom{L - \mathrm{supp}(y)}{k}$ to $T(Y, B_z)$. This is a decreasing function of $\mathrm{supp}(y)$ which implies that $T(Y, B_z)$ is maximized when $Y = A$ and each $y \in Y$ has $\mathrm{wt}(y) = \hat{w}$. ■

Finally, Theorem III.1 follows from this lemma and the fact that at least one hash function must have at most the average number of input vectors in the target bucket.