PETCAM–A Power Efficient TCAM For Forwarding Tables *

Tania Mishra and Sartaj Sahni
Department of Computer and Information Science and Engineering,
University of Florida, Gainesville, FL 32611
{tmishra, sahni}@cise.ufl.edu

November 10, 2008

Abstract

We investigate various TCAM architectures recently proposed for TCAM power and memory reduction and show that far better power and memory performance is possible when we use an optimal prefix set for the given router table than when the original prefix set or the reduced prefix set as proposed in other work is used. For EaseCam [8, 9], our experiments show a power and TCAM memory reduction of 96% to 98% and 62% to 69% respectively. For the suffix node architecture of [3], we get a power and TCAM memory reduction of 16% to 25% and 45% to 78% respectively.

Keywords
Packet forwarding, TCAM, power.

1 Introduction

Internet packets get from source to destination via a number of hops. At each hop, a forwarding engine uses the destination address of the packet and a set of rules to determine the next hop for the packet. A packet forwarding rule (P, H) comprises a prefix P and a next hop H. A packet with destination address d is forwarded to H where H is the next hop associated with the rule that has the longest prefix that matches d (we assume, throughout this paper, that no two rules have the same prefix). We refer to the set of rules as the rule table or router table. Figure 1 shows a small router table with 6 prefixes. The prefix associated with rule R4 is 01 (the * at the end indicates a sequence of don't care bits) and the associated next hop is H4. Rule R4 matches all destination addresses that begin with 01. The length of the prefix 01 associated with R4 is 2. A destination address that begins with 010 is matched by rules R1, R2, R4, and R5. Of these rules, R5 is the one with the longest prefix. So, H5 is the next hop for packets with a destination address that begins with 010.

[11, 12] survey the many solutions that have been proposed for longest prefix matching in the context of packet forwarding. Our focus, in this paper, is longest prefix matching using a TCAM (ternary content addressable memory). Each bit of a TCAM may be set to one of the 3 states 0, 1, and x (don't care). A simple and fast solution to longest prefix matching results from the use of a TCAM in conjunction with an SRAM. The prefix of a rule is stored in a word of TCAM and the next hop is stored in the corresponding SRAM word. Figure 2 shows a TCAM in which each word is 4 bits long; the prefixes of our 6-rule example of Figure 1 have been stored in the

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

	Prefixes	Next Hop
R1	*	H1
R2	0*	H2
R3	1*	H3
R4	01*	H4
R5	010*	H5
R6	111*	H6

Figure 1: An example 6-prefix forwarding table

TCAM in decreasing order of length along with an SRAM in which the next hop information has been stored. A TCAM searches all its words, in parallel, for the first word that matches the content of its search register. By loading a destination address into the search register of a TCAM we can determine the index of the first TCAM word that matches this destination address. Using this index, we then access the corresponding SRAM word to determine the next hop. So, when router-table prefixes are stored in a TCAM in decreasing order of length, we can determine the next hop in 1 TCAM cycle! We note that, in practice, using the described strategy, a TCAM word will be 32 bits for IPv4 applications.

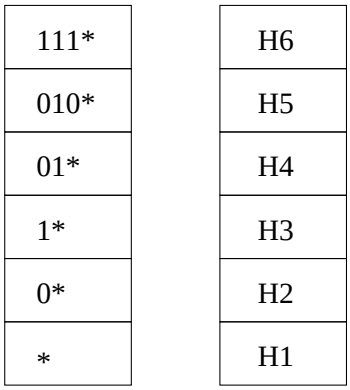


Figure 2: TCAM for the 6 rules of Figure 1

Although TCAMs lead to a very simple and fast solution to the packet forwarding problem of finding the next hop associated with the longest matching prefix, there are several pitfalls associated with their use. These pitfalls include high power consumption, limited capacity, and high cost. Several researchers have recently proposed methods to alleviate the power consumption and capacity limitations. Central to the proposed methods [8, 9, 3, 19] to reduce power consumption is the observation that the power consumed by a TCAM search is proportional to the size of the portion of the TCAM that needs to be searched rather than to the TCAM's overall size. Zane et al. [19] propose a two-level architecture in which the first level extracts some number of bits from the destination address and these extracted bits are used to index into a segment of the TCAM that is to be searched for the longest matching prefix. Ravikumar et al. [8, 9] propose a similar two-level architecture. However, the extracted bits are restricted to be a prefix of the destination address (first 8 bits) and the TCAM segments are of variable

size. The use of variable size segments requires the use of a table of segment start addresses but reduces wasted TCAM space. The two-level schemes of [8, 9, 19] also increase the effective capacity of a TCAM as the word size of the TCAM is reduced by the number of extracted bits. So, in an IPv4 application, for example, if 8 bits are used for the first-level indexing, a TCAM word need only be 24 bits rather than 32 bits (as in the scheme of Figure 3). Liu [7] proposes the use of pruning and mask extension to compact a TCAM table and hence reduce the number of rules that has to be stored. This compaction reduces power consumption and also increases the effective capacity of the TCAM. Lu and Sahni [3] propose table segmenting methods and the use of wide SRAMs to reduce power consumption and increase effective table capacity.

In this paper, we propose the use of a minimum set of rules equivalent to those in the given router table coupled with the wide SRAM strategy of [3]. We perform batch updates to the set of rules to accomodate the incoming route advertisements. We begin in Section 2 by reviewing related work. In this section, we clarify the proposal of [7] and point out deficiencies in the scheme of [8, 9]. In Section 5 we describe our proposed PETCAM method. An experimental evaluation of the various methods proposed for low-power TCAMs is done in Section 6.

2 Background and Related Work

Much research has been done to improve the power efficiency of TCAM-based router tables [7, 3, 8, 9, 19, 13, 14, 15, 16, 17]. Pure hardware approaches for power reduction are presented in [13, 14, 15, 16]. Z. Wang et al in [17] present an algorithm for consistent and incremental updates to TCAMs. We describe the results reported in [7, 3, 8, 9, 19] in this section as these are most relevant to the work we report in this paper.

Definition 1 *$P1 \subset P2$ iff $addr(P1) \subset addr(P2)$, where $addr(P)$ is the set of addresses matched by prefix P. Note that $P1 \subset P2$ iff $P2$ is a proper prefix of $P1$.*

Definition 2 *A rule $(P1, H1)$ is* Type I redundant *iff (a) there exists a rule $(P2, H2)$ such that $P1 \subset P2$ and $H1 = H2$ and (b) there is no rule $(P3, H3)$ such that $P1 \subset P3 \subset P2$.*

Definition 3 *A* generalized prefix *is a sequence comprised of the symbols 0, 1, and ? and possibly terminated by the symbol *. A* simple prefix *(or simply, prefix) is a generalized prefix that has no occurrence of the symbol ?. (Alternatively, we may limit the occurrence of the symbol ? to the right end of the sequence. Note that ?s at the right end of a sequence may be replaced by a * so that the sequence 10??? may be regarded as a simple prefix by rewriting it is 10*.)*

For example, 0??1??0* and ??100?11* are generalized prefixes. In router table applications, a generalized prefix may be stored in a word of TCAM by replacing * with a suitable number of ?s.

Definition 4 *Two sets of generalized prefixes are* equivalent *iff they match the same addresses.*

Liu [7] proposes two schemes–pruning and mask extension–to compact the rules of a router table. In pruning, rules with type I redundant prefixes are eliminated from the rule table. It is easy to see that the elimination

of type I redundant prefixes does not change the next-hop decision for any destination address. Following the elimination of type I redundant prefixes, each set, S, of prefixes that have the same length and the same next hop is subjected to *mask extension* in which S is replaced by an equivalent set of generalized prefixes T such that $|T| \leq |S|$. Liu [7] proposes the use of a logic minimization heuristic–Espresso II–to compute a nearly minimal equivalent set T. Liu [7], however, does not address the issue of how to assign lengths to the generalized prefixes of T (or, equivalently, how to place the generalized prefixes of T into the TCAM) so that a TCAM search reports the same next hop as reported using longest prefix matching on the original set of simple prefixes. We address this issue in Section 3. Liu [7] reports that pruning and mask extension result in a reduction of 42% to 48% in the number of generalized prefixes that need to be stored in the TCAM. Note that without pruning and mask extension, we store simple prefixes in the TCAM. However, the TCAM word size is the same regardless of whether simple or generalized prefixes are stored. So, a 42% reduction (say) in the number of generalized prefixes translates to a 42% reduction in TCAM memory.

Ravikumar et al. [8, 9] extend the work of Liu [7] and propose the 2-level EaseCAM architecture for router tables (Figure 3). For an IPv4 router table, the first level stores 8-bit sub-prefixes. Prefixes that have the same first 8 bits define a prefix cluster. Pruning, prefix aggregation, and prefix expansion are used to replace the simple prefixes in each cluster with a smaller set of generalized prefixes with the property that a search of the TCAM segment that contains this smaller set of generalized prefixes results in the same next hop as does a search in the TCAM segment for the original cluster of simple prefixes. Since the generalized prefixes in a cluster have the same first 8 bits, it is necessary to store only the remaining 24 bits of each generalized prefix in the second-level TCAM (note that to store the terminating * of a generalized prefix, we must replace it with a sufficient number of ?s so that the total number of symbols in the generalized prefix is 32). Consequently, second-level TCAM words are 25% smaller than the TCAM words in the design of [7]. Prefixes shorter than 8 bits are stored in a separate bucket. The pruning process of Ravikumar et al. [8, 9] is identical to that of Liu [7]–type I redundant prefixes are eliminated. The compaction process of Ravikumar et al. [8, 9] differs from that of Liu [7] in how generalized prefixes are created from a set of same-hop prefixes that is free of type I redundancies. In an effort to reduce the time required by Espresso to process same length same hop prefixes, Ravikumar et al. [8, 9] propose aggregating prefixes (in a cluster) that have the same hop into sets in which the prefixes have a common longest sub-prefix of size a multiple of 8. Then, prefixes in each such aggregated set are expanded using prefix expansion [8, 9] so that the length of each prefix is a multiple of 8. For example, following aggregation the prefixes in an aggregated set may have length between 16 and 23 with all prefixes in this set having the same first 16 bits. Using prefix expansion, the lengths of all prefixes in the set becomes 24. Since all prefixes in this prefix-expanded aggregated set have the same first 16 bits, Espresso may be used to find a minimum number of generalized prefixes equivalent to the 7-bit suffixes in this set. Working with 7-bit suffixes rather than full 23-bit prefixes reduces the run time of Espresso [9]. The fact that the aggregated prefix-expanded sets are relatively small (compared to sets of same hop same length prefixes) is another (and significant) contributing factor to the observed reduction in time spent on Espresso optimization. Although prefix aggregation and expansion reduce Espresso time with little loss in

compaction effectiveness [8, 9], there are correctness issues that we address in Section 4. Since the power consumed by a TCAM lookup is proportional to the size of the TCAM segment that is searched rather than to the overall TCAM size, the scheme of [9] achieves power reduction from using a compacted set of prefixes, storing only the last 24 bits of each prefix in TCAM, and from searching only the prefixes in a cluster.

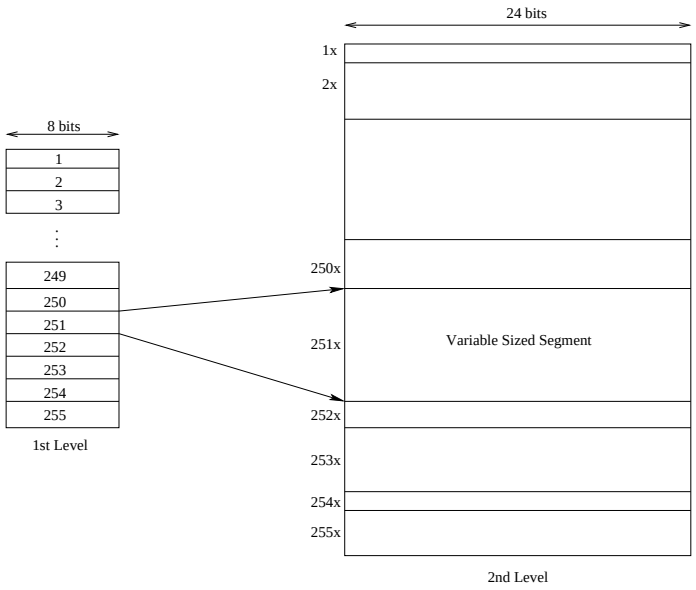


Figure 3: EaseCam architecture of [9]

Zane et al. [19] propose two schemes to achieve power reduction. In the first, bit selection, a few bits (not necessarily the first few) of each prefix are used to partition the prefix set so that each partition agrees on these selected bits. The bits are called the *partition selector bits*. Prefixes in the same partition are stored together in decreasing order of length. To search for the longest matching prefix for a given destination address d, the partition selector bits are extracted from d and used to determine which partition is to be searched. Although all prefixes of an uncompacted router table are stored in the TCAM, power reduction results from having to search only one partition [1]. Additional power reduction is possible if the partition selector bits are extracted from the prefixes before storage in the TCAM as this results in a reduction in the total number of bits in a partition. Note that bit selection, which predates the work of [9], is similar to the 2-level strategy employed in [9], where the first 8 bits are used to determine the partition to search.

The second strategy proposed by Zane et al. [19] is a 2-level TCAM architecture in which the first level TCAM is an index to the partitions in the second level TCAM. The partitions and index are constructed by decomposing the binary trie representation of the router-table prefixes. Although both Zane et al. [19] and Ravikumar et al. [9] propose 2-level TCAM architectures, Zane et al. [19] do not compact the router table (except when bit selection is used and the partition selector bits are not stored in the second level TCAM) while Ravikumar et al. [9] do. As a result, the total TCAM memory required by the schemes of Zane et al. [19] is more than that required by the

[1] The power required by a TCAM lookup is proportional to the total number of bits in the TCAM partition that is searched.

scheme of Ravikumar et al. [9].

The most recent work on TCAM power reduction in the context of router tables appears to be that of Lu and Sahni [3]. They augment the traditional 1-level TCAM lookup structure as well as the 2-level TCAM structure of Zane et al. [19] with wide SRAMs and store the suffixes of several prefixes in a single wide SRAM word. This enables a reduction in both power consumption and total TCAM memory requirement.

3 Issues Related to [7]

As noted in Section 2, when logic minimization is applied to a set of same-hop same-length prefixes, we get a set of equivalent generalized prefixes. So, for example, $A = \{000*, 001*, 010*, 011*\}$ optimizes to $B = \{0*\}$. While it may be natural to assign 0* a length of 1, such a length assignment can result in an incorrect next hop computation. To see this, suppose that the next hop associated with the prefixes of A is $H1$ and that the router table has another prefix $00*$ whose next hop is $H2$ and $H1 \neq H2$. When using the original prefix set $C = \{000*, 001*, 010*, 011*, 00*\}$, packets with destination address beginning with 000 are sent to $H1$. Consider what happens when we apply the compaction scheme of Liu [7]. Since C has no type I redundancy, pruning does not weed out any member of C. Mask extension compacts A to B. So, the compacted prefix set is $D = \{0*, 00*\}$ with 0* having $H1$ as its next hop and 00* having $H2$. Using the prefix set D, packets with destination addresses that begin with 000 are sent to $H2$! We can overcome this difficulty in one of two ways. The first and simplest is to declare the length of each generalized prefix in the optimized set D to be the same as that of the prefixes in the set A. This ensures that, when prefixes are loaded to the TCAM in length order, the outcome is the same (in terms of next hop) as when the original prefix set is loaded in length order. For example, using this definition of length for a generalized prefix, 0* in set D has length 3, and prefix 00* has length 2. Thus, 0* is loaded first in the TCAM followed by 00*.

The second strategy is to use a more intuitive definition of length such as the index of the rightmost symbol that is not a ? or a *. So, the length of 1??01* is 5 and the length of ??00??1* is 7. This is consistent with the accepted definition of the length of a simple prefix where, for example, the length of 001* is 3. We use the notation $|G|$ to denote the length (using the just stated intuitive definition) of the generalized prefix G. Using such a definition works provided we remove also type II redundant rules as is shown below.

Definition 5 *A rule* (P_1, H_1) *is* Type II redundant *iff the router table contains a set of rules* $\{(P_2, H_2), \cdots, (P_k, H_k)\}$ *such that* $|P_1| < |P_i|$, $2 \leq i \leq k$ *and every address matched by* P_1 *is also matched by a* P_i, $2 \leq i \leq k$.

In the rule set {(10*,H1), (100*,H2),(101*,H3)}, no prefix is type I redundant. However, (10*,H1) is type II redundant. Neither Liu [7] nor Ravikumar et al [8, 9] remove type II redundant rules. We note that every generalized prefix may be written as the sum of simple prefixes that have the same length as the generalized prefix and such that the addresses matched by the generalized prefix are the union of those matched by the simple prefixes. So, for example, 1?00?1* = 100001* + 100011* + 110001* + 110011*. This decomposition of a generalized prefix

into the sum of simple prefixes that have the same length as the generalized prefix is referred to as *generalized prefix decomposition* (GPD) and $GPD(X)$ is the generalized prefix decomposition of the generalized prefix X.

Definition 6 *Let $R = \{R_1, R_2, \cdots, R_r\}$ be a set of generalized prefixes that is equivalent to the set of simple equal-length same-hop prefixes $S = \{S_1, \cdots, S_s\}$. R is a* canonical *equivalent set iff each R_i is the sum of some of the S_qs.*

Theorem 1 *Let R and S be as in Definition 6. There exists a canonical equivalent set for S that has the same number of generalized prefixes as does R.*

Proof Consider an R_i in R. Let $R_i = R_{i1} + R_{i2} + \cdots + R_{iq(i)}$ be the GPD of R_i. Since R and S are equivalent and prefixes of the same length are disjoint (i.e., have no common matching address), there is exactly one $f(i,j)$, $1 \le f(i,j) \le s$, such that R_{ij} and $S_{f(i,j)}$ are not disjoint, $1 \le i \le r$, $1 \le j \le q(i)$. We consider 3 cases.

Case 1: If $|R_i| = |S_1|$, $R_{ij} = S_{f(i,j)}$ for all j and so R_i is the sum of some of the S_qs.

Case 2: If $|R_i| > |S_1|$, let R_i^* be the first $|S_1|$ bits of R_i. So, the addresses matched by R_i are a subset of those matched by $R_i^* = R_{i1}^* + R_{i2}^* + \cdots + R_{iq(i)}^* = S_{f(i,1)} + S_{f(i,2)} + \cdots + S_{f(i,q(i))}$, where R_{ij}^* is obtained from R_{ij} by truncating the last $|R_i| - |S_1|$ bits. Since R_i^* matches no address not matched by S, replacing R_i by R_i^* in R preserves the equivalence between R and S and doesn't increase the number of R_is in R. We may use this replacement transformation as often as need to replace all R_is in R whose length is more than $|S_1|$ with R_i^*s whose length equals $|S_1|$. From Case 1, it follows that each of the replacing R_i^*s is the sum of some of the S_qs.

Case 3: When $|R_i| < |S_1|$, we may use prefix expansion to represent each R_{ij} as the sum of 2^t, $t = |S_1| - |R_i|$ simple prefixes whose length is $|S_1|$. From the equivalence of R and S and the fact that prefixes of the same length are disjoint, it follows that each expanded prefix is one of the S_qs. So, each R_{ij} and hence R_i is the sum of some of the S_qs.

■

The prefixes of a canonical equivalent set are called *canonical prefixes* and $CD(R_{ij})$ is the set of prefixes of S that sum to R_{ij}. From Theorem 1, it follows that for every set of equivalent generalized prefixes computed by a minimization algorithm, there is a canonical equivalent set with the same number of generalized prefixes. So, henceforth, we assume that minimization algorithms return canonical prefixes.

Theorem 2 *Let U be a set of rules comprised of simple prefixes that is free of type II redundancies. Let V be the set of rules comprised of (canonical) generalized prefixes obtained from U by applying logic minimization to the equal-length same-hop prefixes of U as is done in mask extension [7]. Longest prefix matching in U and V results in the same next hop for every destination address A.*

Proof Suppose there is an address A for which the longest matching simple prefix in U is U_1 with next hop H_1 and for which the longest matching generalized prefix in V is V_2 with next hop H_2 and $H_1 \neq H_2$. Let V_{21} be the prefix of $GPD(V_2)$ that matches A. Note that since all prefixes in $GPD(V_2)$ have the same length, they are disjoint and so exactly one of these matches A. Further, let U_2 be the prefix of $CD(V_{21})$ that matches A. Again, exactly one prefix of $CD(V_{21})$ matches A. Since U_1 is the longest prefix of U that matches A, $|U_1| > |U_2|$. Let V_1 be the generalized prefix of V such that $V_{11} \in GPD(V_1)$ matches A and $U_1 \in CD(V_{11})$. Such a V_1 must exist in V because of the way V is constructed from U using logic minimization. Since V_2 is the longest matching generalized prefix for A in V and V_1 also matches A, $|V_{21}| = |V_2| \geq |V_1| = |V_{11}|$. Now, since two prefixes are either disjoint or nest and since U_1, U_2, V_{11}, and V_{21} match A,

$$addr(U_1) \subset addr(U_2) \subseteq addr(V_{21}) \subseteq addr(V_{11})$$

From this and the observation that all prefixes in $CD(V_{11})$ are of the same length and hence are disjoint, it follows that some subset of $CD(V_{11})$ that includes U_1 sums to U_2. Hence, U_2 is type II redundant. ■

From Theorem 2, it follows that if we start with a set of prefixes that contains no type II redundancy, apply the reductions of [7] to obtain generalized prefixes, and enter these generalized prefixes into a TCAM in decreasing order of length, then lookups yield the same next hops as when we load the TCAM with the non-reduced prefix set in length order.

4 Issues Related to [8, 9]

The issues with the mask extension method of Liu [7] may be resolved by either using an unnatural definition for the length of a generalized prefix (i.e., length equals that of the equal-length simple prefixes that were input to the logic minimizer) or by eliminating type II redundancies prior to logic minimization and defining length as in the definition of $|G|$ provided in Section 3. These resolution methods do not, however, extend to the aggregation and prefix expansion techniques proposed in [8, 9] to reduce the number of rules to be stored in the TCAM.

4.1 Prefix Aggregation

In prefix aggregation, prefixes that have the same hop are aggregated into clusters with each cluster containing prefixes that have the same common sub-prefix. The common sub-prefix length is constrained to be a multiple of 8. So, for example if two prefixes that have the same next hop agree on their first 18 bits only, then they will be in a cluster of same-hop prefixes that agree on their first 16 bits. Logic minimization is then applied to each cluster. Since the prefixes in a cluster have different length, there appears to be no reasonable way to determine where to place the generalized prefixes that result from logic minimization into the TCAM so as to correctly route packets. Neither of the length resolution methods proposed for mask extension in Section 3 work when aggregation is employed. For example, consider the rule set {(1*,A), (10*,B), 101*,A)}, where the first 8 bits of each prefix are omitted and are the same. The rule set is devoid of type I and type II redundancies and so no rule is eliminated in

the initial pruning step. In the aggregation step, 1* and 101* form a cluster and 10* is in a different cluster as it has a different next hop. Logic minimization reduces the first cluster to 1* and has no effect on the second cluster. The new rule set is {(1*,A), (10*,B)}. 1* was derived from a prefix of length 1 and one of length 3. Neither length assignment 1 or 3 for 1* allows the new rule set to work like the original rule set. For example, with the natural length assignment of 1 to 1*, packets destined to 101* addresses get routed to B rather than to A and with a length assignment of 3, packets to 10* get sent to A rather than to B!

4.2 Prefix Expansion

[8, 9] propose using prefix expansion within an aggregated cluster to improve the runtime performance of logic minimization. In prefix expansion, short prefixes in a cluster are replaced by a set of prefixes whose length equals that of the longest prefix in the cluster. So, following prefix expansion, all prefixes in a cluster have the same length. Since logic minimization is faster when the input prefixes are of the same size, runtime efficiency is achieved [8, 9]. In the example cluster {1*,101*} of Section 4.1, prefix expansion yields the cluster {100*, 101*, 110*, 111*}, which is reduced to 1* by logic minimization. The new rule set is {(1*,A),(10*,B)}, which, as noted in Section 4.1 cannot be made to work the same as the original rule set.

5 PETCAM

Our power-efficient TCAM, PETCAM, employs the following construction steps:

Step 1: Transform the given routing table to an equivalent optimal routing table using the dynamic programming algorithm of [10].

Step 2: Use mask extension as in [7] to reduce the number of prefixes in the optimal routing table obtained in Step 1 even further. This is possible as the optimal routing table is limited to be comprised of simple prefixes alone whereas mask extension results in generalized prefixes.

Step 3: Map the reduced set of generalized prefixes constructed in Step 2 to a 2-level TCAM augmented with a wide SRAM by extending the suffix node method developed in [3].

Since the dynamic programming algorithm of [10] transforms a set of prefix rules into a provably optimal equivalent set of prefix rules, the transformed set is guaranteed to be free of type I and type II redundancies. Hence, the generalized prefixes that result from the mask extension done in step 2 correctly classify packets when these prefixes are entered into a TCAM in decreasing order of length ($|G|$). For step 3, we need to adapt the suffix-node method of [3] so as to accommodate generalized prefixes rather than simple prefixes. For this adaptation, we need to modify the structure of a suffix node as well as develop an algorithm to map suffixes into suffix nodes. Before developing these adaptations, we provide a brief overview of the suffix-node method of [3].

5.1 Suffix-Node Method of [3]

Lu and Sahni [3] propose the use of wide SRAMs in conjunction with TCAMs so as to reduce power consumption and increase effective TCAM capacity. Although Lu and Sahni [3] propose methods for both 1- and 2-level TCAMs, we review only the 1-level method here and adapt this to generalized prefixes. A similar adaptation may be done for the 2-level methods of [3].

Lu and Sahni [3] make the observation that the simple TCAM organization of Figure 2 does not make effective use of modern wide access SRAMs. Whereas a next hop can often be encoded using 10 to 12 bits we can fetch, in a single memory fetch cycle, 72 bits from a QDRII SRAM in dual burst mode and 144 bits in quad burst mode. A larger number of bits may be fetched per cycle by employing multiple SRAMs that may be simultaneously accessed. Further, given the orders of magnitude discrepancy between the time for an SRAM fetch cycle and the time to perform an arithmetic, it is possible to do significant processing of the data stored in a word of a wide SRAM in much less time than it takes to fetch that word of data from the SRAM. To capitalize on these observations, Lu and Sahni [3] propose packing the suffixes of several router-table prefixes that are in the same subtree of the binary trie for the router-table prefixes into a suffix node, which is then stored in one or more SRAM words in such a way that the entire suffix node may be retrieved in a single memory cycle. Figure 4 gives the structure of the suffix node of [3]. We have added a 5-bit match start position field which indicate the bit position in the destination prefix from where suffix matching can start for all suffixes encoded in the suffix node. The suffix count field gives the number of suffixes packed in the suffix node. For each suffix Si stored in a suffix node, we keep the suffix length, $len(Si)$, the suffix, Si, and the next hop associated with the suffix. Using the suffix node creation scheme in [3], each suffix node must have exactly one suffix of length 0. This suffix can come from either a prefix that is stored in the root of the subtree that is carved to form the suffix node, or a covering prefix which is inherited from the nearest ancestor with a prefix in case the root of the subtree does not store a prefix. To optimize SRAM further, we store this suffix as the first suffix in the node, and since it has a length 0, we drop the suffix length field for the first suffix. Thus a suffix of length 0 appears as the first suffix in a suffix node, and is represented only by its next hop.

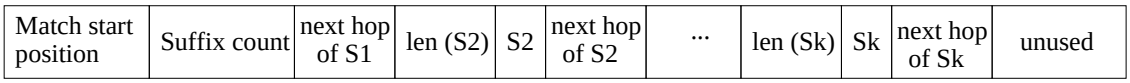


Figure 4: Suffix node of [3] with a 5-bit match start position field and an optimized representation of the first suffix.

Figure 5 shows the binary trie for the prefixes of Figure 1 together with a mapping of these prefixes into a simple TCAM with wide SRAM. For this example, we assume an SRAM word width of 32 bits with 2 bits allocated for the match start position field (allowing prefixes to be of length 5 bits), 2 bits allocated for the count field of a suffix node (permitting up to 4 suffixes to be stored in a node), 2 bits for the suffix length field (permitting suffixes of length between 0 and 3), and 12 bits for the next-hop field (permitting upto 4096 different next hops). In the worst-case, a suffix node stores a single suffix of length 0, for which a next hop field of 12 bits is used along with the

match start position and suffix count fields, utilizing only 16 of the 32 bits. In the best case, we may store a suffix of length 0 and one of length 2 resulting in the utilization of all 32 bits in the node. The allocation of suffixes to suffix nodes is done by carving out subtries of the binary trie for the prefix set. For example, from the binary trie of Figure 5, we first carve out the binary trie rooted at node A. The path from the root to A is $Q(A) = 01$. $Q(A)$ is stored in the TCAM and the suffixes * (length 0) and 0* (length 1) that result from eliminating $Q(A)$ from the front of each prefix in the carved subtrie are packed into a suffix node. This carving-packing process is repeated at nodes B, C, and D resulting in the suffix nodes of Figure 5. When carving is done at a node R whose subtree doesn't contain a matching prefix for every destination address that begins with $Q(R)$, we add a *covering prefix* into the suffix node for this carving. The covering prefix for node R, which is stored as a suffix whose length is 0 is the prefix in the nearest binary trie ancestor of R. Assuming that each router table contains the default prefix *, each node of the binary trie has a well defined covering prefix. The covering prefix for node B of the binary trie of Figure 5 is P3 with a next hop of H3. In practice, we store a covering prefix whenever the root of the carved subtree does not contain a prefix. Hence, every suffix node has a prefix, its first one, whose length is 0.

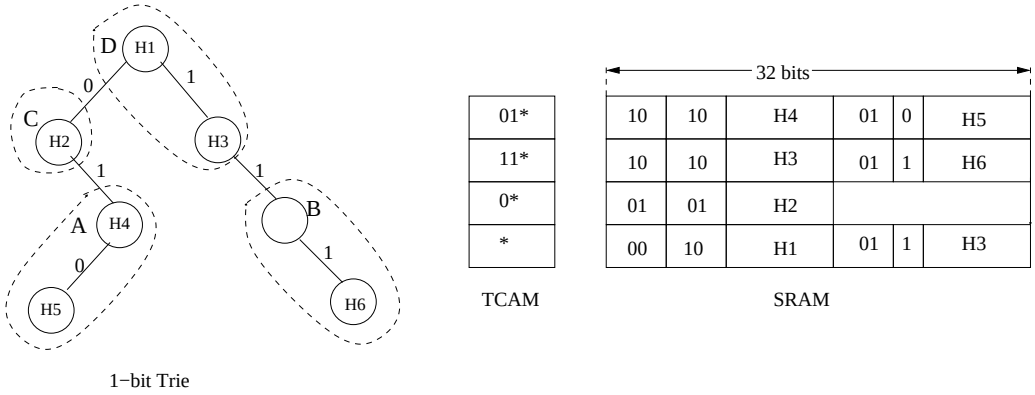


Figure 5: Suffix node example

The $Q(R)$s and associated suffix nodes are assigned, respectively, to TCAM and SRAM words in descending order of length [3].

5.2 Our Suffix-Node Structure

The suffix node method of [3] cannot be used as is for PETCAMs because, in a PETCAM, we store generalized prefixes rather than simple prefixes. Specifically, we need to define a new format for a suffix node as well as formulate an algorithm to populate suffix nodes with the suffixes of generalized prefixes. Our new suffix-node structure has a 1-bit type field that permits the use of two variants. A type I suffix node is used to store simple suffixes exclusively (i.e., all suffixes in a type I suffix node are comprised of 0s and 1s). Such a suffix node is structured the same as the suffix node of [3] except for the addition of a type field (Figure 6).

A type II suffix node (Figure 7) stores a mix of simple and non-simple suffixes (i.e., suffixes that have at least one don't care bit). Simple suffixes are stored first using triples (length, suffix, next hop) as used in Figure 6. These

5bits	4 bits	1 bit	4 bits		12 bits		4 bits		12 bits	
Match start position	Suffix count	Type	length(S1)	S1	next hop of S1	...	length(Sk)	Sk	next hop of Sk	unused

Figure 6: Type I suffix node

triples are followed by 4-tuples (length, suffix, mask, next hop) that represent non-simple suffixes. The suffix and mask entries are of the same length and the 1s in the mask identify the don't cares in the suffix. For example, the suffix x0x1 may be represented by the simple suffix 0001 and the mask 1010, for example. The *index* field gives the index of the first non-simple suffix. So, for example, if we have 2 simple suffixes and 3 non-simple suffixes in a type II suffix node, the *count* field would be 5 and the *index* field would be 3.

5 bits	4 bits	1 bit	3 bits	12 bits	4 bits			12 bits		4 bits			12 bits	
Match start position	Suffix count	Type	Index	next hop of S1	len(S2)	S2	M2	next hop of S2	...	len(Sk)	Sk	Mk	next hop of Sk	unused

Figure 7: Type II suffix node

5.3 Normalized Ternary Tries

To map the generalized prefixes that result from steps 1 and 2 of our PETCAM construction algorithm we first construct a ternary trie[2]. Figure 8 shows an example router table following steps 1 and 2 of our PETCAM construction algorithm. Figure 9 shows the corresponding ternary trie.

	Address	Next Hop
1	x0	H1
2	00x0	H2
3	00x1	H3
4	1100	H4
5	11x1	H5

Figure 8: Router table with generalized prefixes

A *normalized* ternary trie is a ternary trie in which each node that is the x-child (i.e., the don't care child) of its parent has no sibling. So, in a normalized ternary trie, the children of degree 2 nodes are 0- and 1-children, the child of a degree 1 node may be a 0-, 1-, or x-child, and there are no degree 3 nodes. A ternary trie may be normalized by eliminating the x-child of each degree 3 node by merging the subtree rooted at this x-child with the subtrees rooted at the two siblings of this x-child. For example, in the ternary trie of Figure 9, the root is a degree 3 node and the subtree rooted at its x-child may be merged with the subtree rooted at the root's 0-child as well as with that rooted at the root's 1-child to obtain the ternary tree of Figure 10. One may verify that the ternary tries of Figures 9 and 10 are equivalent in that both route all packets to the same next hop.

[2]A ternary trie differs from a binary trie in that each node of a ternary trie may have up to 3 children depending on whether the branching bit is a 0, 1, or an x.

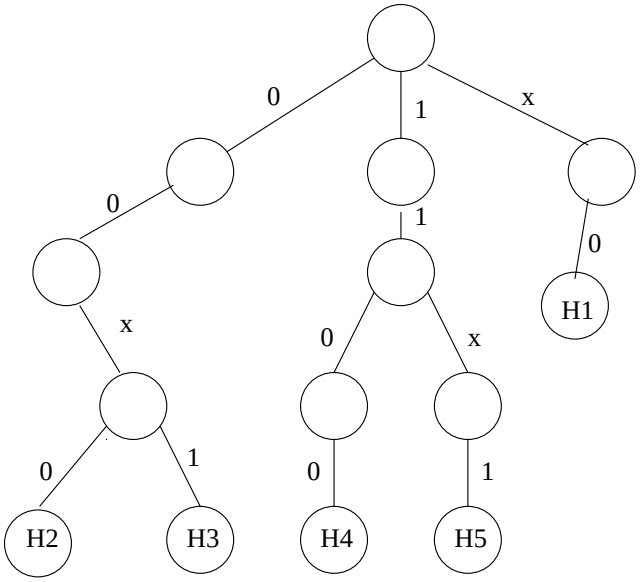


Figure 9: Ternary trie for the router-table of Figure 8

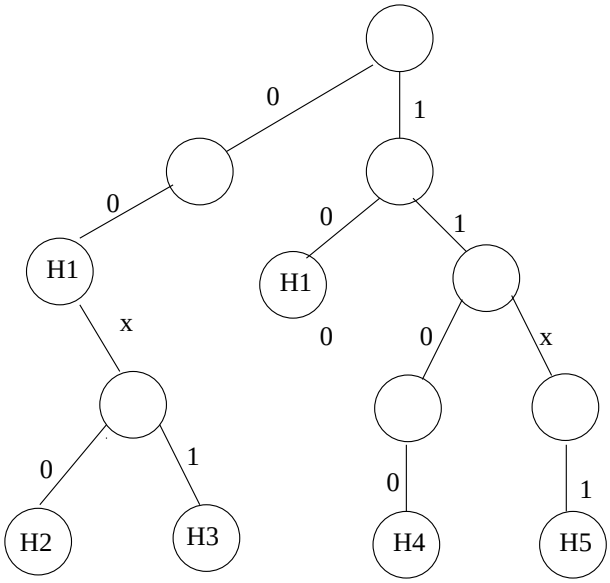


Figure 10: Ternary trie following the merging of the x-child of the root of Figure 9

The trie of Figure 10 is not yet a normalized ternary trie as it contains an x-child that has a sibling (i.e., the x-child with $Q(R) = 11x$). This subtree rooted at this x-child may be merged with that rooted at its 0-sibling and its empty 1-sibling to obtain the normalized ternary trie of Figure 11.

Figure 12 gives our algorithm to normalize a ternary trie. This algorithm assumes that each node y of a ternary trie has 3 children fields with $y.child[0]$ and $y.child[1]$ pointing to the 0- and 1-children of node y and $y.child[2]$ pointing to the x-child of node y. The algorithm employs two other algorithms–*delete*, which deletes a subtree given its root and *merge*, which merges two subtrees together. We do not further specify *delete* as this is a simple

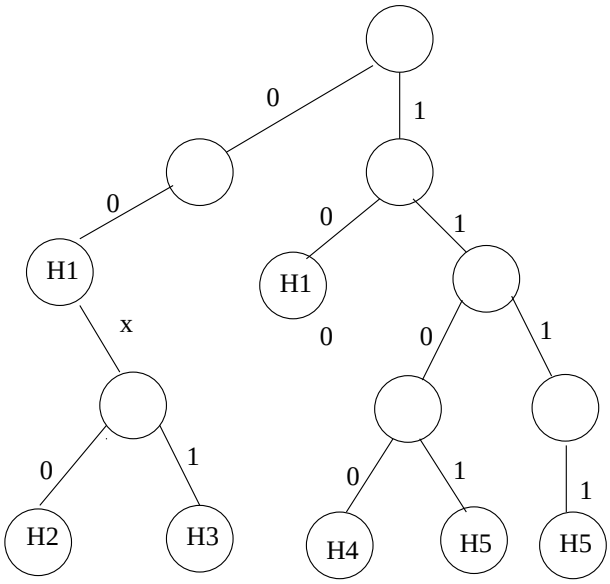


Figure 11: Normalized ternary trie following merging of the $Q(R) = 11x$ subtree of Figure 10

postorder traversal. Algorithm *merge* is specified in Figure 13.

```
Algorithm normalize(root)
{
    if (!root) return;
    if (root.child[2]) {
       if (root.child[0] || root.child[1]) {
          merge (root, root.child[0], 0, root.child[2]);
          merge (root, root.child[1], 1, root.child[2]);
          delete(root.child[2]);
          root.child[2] = NULL;
       }
    }
    normalize(root.child[0]);
    normalize(root.child[1]);
    normalize(root.child[2]);
}
```

Figure 12: Algorithm to normalize a ternary trie

In algorithm *merge*, *oChild* and *xChild* are children of *parent*. *xChild* is the x-child while *oChild* is the *oChildID*-child. Notice that when we start with a prefix set that has no type I and II redundancies and perform mask extension, at most one of *oChild* and *xChild* may have a non-null next hop. Further, note that an optimal prefix set is devoid of type I and type II redundancies.

5.4 Our Carving Heuristic

Our carving heuristic starts with the normalized ternary trie for the canonical prefixes that result when mask extension is done on an optimal prefix set. To carve the normalized ternary trie into suffix nodes, we first compute

```
Algorithm merge(parent, oChild, oChildID, xChild)
{
    if (!xChild) return;
    if(!oChild) {
       oChild = new node;
       oChild.nextHop = xChild.nextHop;
       parent.child[oChildID] = oChild;
    }
    else {
       if (!xChild.nextHop) then
          oChild.nextHop = xChild.nextHop;
    }
    merge (oChild, oChild.child[0], 0, xChild.child[0]);
    merge (oChild, oChild.child[1], 1, xChild.child[1]);
    merge (oChild, oChild.child[2], 2, xChild.child[2]);
}
```

Figure 13: Algorithm to merge an x-subtree

the following values for each node y of the normalized trie.

1. $y.numP \cdots$ number of prefixes stored in the subtrie rooted at y. This is equivalent to the number of nodes in this subtrie that have a non-null next hop field. Let $y.h = 0$ if $y.nextHop$ is null and 1 otherwise. It is easy to see that $y.numP$ is the sum of the $numP$ values for its up to 2 non-empty subtrees plus $y.h$.

2. $y.xNumP \cdots$ number of nodes in the subtree rooted at y that have a non-null next hop stored and the path from y to each of these nodes includes at least one x-child other than y. Note that if y has an x-child it can have no other child and so $y.xNumP$ is the $numP$ value of this x-child. When y does not have an x-child, its $xNumP$ value is the sum of the $xNumP$ values of its children.

3. $y.size \cdots$ number of bits needed to store the suffixes (together with suffix count, node type, index (if required), suffix lengths, masks (if required), and next hops) for the prefixes in the subtree rooted at y. Each such suffix is obtained by removing $Q(p)$ from the $y.numP$ prefixes in the subtree rooted at y. In case $y.xNumP = 0$, a type I suffix node is used. Otherwise, a type II suffix node is used. $y.size$ also includes the bits needed to store the next hop for the covering prefix for y in case this is needed. When a covering prefix is needed, we store a suffix of length 0 along with the next hop associated with this covering prefix.

Figure 14 gives the $numP$, $xNumP$, and $size$ values for each of the nodes of the normalized ternary trie of Figure 11, where $size$ here includes only a nexthop and suffix bits for simplicity.

To carve a normalized ternary trie into suffix nodes that use at most w bits per node, we perform a postorder traversal of the trie using the visit function of Figure 15, which differs from that of [3] in the manner in which $size$ is computed. Although the visit function, as stated in Figure 15, does not make explicit use of $numP$ and $xNumP$, these values are useful in the computation of $size$. Note that in Figure 15, the value of $y.size$ is its value at the time y is visited and accounts for the fact that several of y's original subtrees may have been carved out by this time.

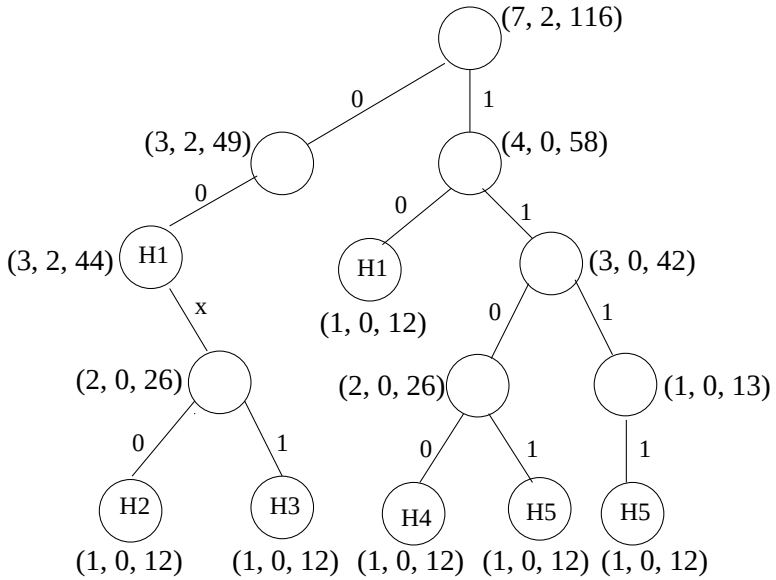


Figure 14: ($numP$, $xNumP$, $size$) for nodes of normalized ternary of Figure 11. Nodes that require a covering prefix are labeled *

```
Algorithm visit(y)

if (y.size < w) return;
if (y.size == w) {carve(y); return;}
// y.size > w
if (y has a single child z) {carve(z); return;}
       // z could be 0-, 1- or x-child

// y has both a 0-child u and a 1-child v
if (u.size <= v.size) {
  carve (v);
  recompute y.size;
  if (y.size < w) return;
  if (y.size == w) carve(y);
  else carve(u);
}
else // u.size > v.size
this is symmetric to the case u.size <= v.size
```

Figure 15: Visit function for subtree carving heuristic [3]

5.5 PETCAM Updates

PETCAM supports batch updates rather than incremental updates. To support batch updates, we use two copies of the TCAM-SRAM lookup subsystems as shown in Figure 16. At any given time, one copy of the TCAM-SRAM subsystem is used for lookup and the other is used to prepare an updated version of the lookup table. In a batch update, the control plane processes all updates received. This is done using a control plane representation (e.g., a trie) of the routing table. With some frequency, the PETCAM construction algorithm is run, creating an updated

TCAM-SRAM representation is the subsystem not currently used for lookup. When construction is complete, lookup is switched to the new subsystem. So, lookups have minimal interruption; the interruption time being that to switch from one subsystem to another. For this strategy to work, the delay between successive rebuilds must at least equal the time to run the PETCAM construction algorithm. Hence, it is important to have an efficient PETCAM construction algorithm.

The same strategy may be used for batch updates using the TCAM schemes of [3, 7, 8, 9]. We note that none of these schemes provide efficient support for incremental updates.

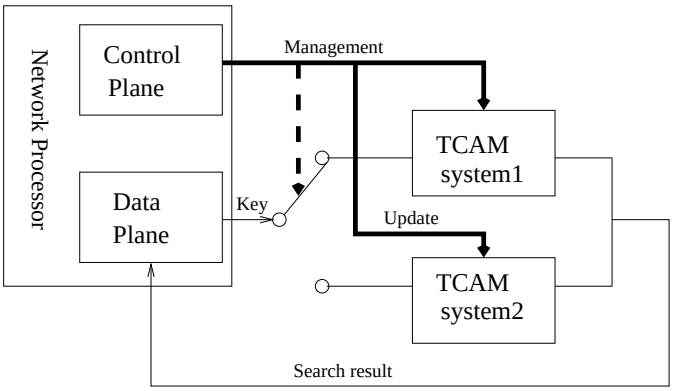


Figure 16: Router architecture using PETCAM.

6 Experimental Results

We programmed our PETCAM strategy in C++ and compared its performance with the power reduction schemes of [7, 3, 8, 9]. The comparison was done using the IPv4 router tables AS1221, AS4637, AS6447, and AS65000, which were obtained from [5] and rrc00, which was obtained from [6]. Data sets AS65000 and rrc00 are from May 2008, AS6447 is from July 2008, and the remaining data sets are earlier than 2008 and were used in [3], for example. Our experiments aim to measure the relative effectiveness of the scheme of Liu [7] (type I redundancy removal followed by mask extension) and the PETCAM scheme (dynamic programming optimization followed by mask extension) to compact the router table as well the overall relative performance of PETCAM, EaseCam, and the method of Lu and Sahni [3] with respect to TCAM power and memory reduction. For our experiments we assume the SRAM word size, and hence the size of a suffix node, is 144 bits.

6.1 Compaction Efficiency

The compaction efficiency is measured by the number of prefixes following the compaction steps. Figure 17 gives the number of prefixes in each of our data sets as well as the number of prefixes following each of steps 1 and 2 of the PETCAM strategy. The dynamic programming algorithm of [10] reduces the number of prefixes in the data sets by between 45% and 79%. Another approximately 5% reduction is achieved when mask extension is employed

on the optimal prefixes produced by the algorithm of [10]. So, PETCAM reduces the number of prefixes by about 50% to 84%.

DataSet	initial # of prefixes	# after Step 1	Reduction (%)	# after Steps 1 and 2	Total reduction (%)
AS1221	281516	153885	45.34	135879	51.73
AS4637	210119	43368	79.36	32562	84.50
AS6447	275509	149117	45.88	137151	50.22
AS65000	259026	81341	68.60	66808	74.21
rrc00	266185	92239	65.35	83146	68.76

Figure 17: Number of router table prefixes in PETCAM

Figure 18 gives the number of prefixes following the removal of type I redundant prefixes as well as following mask extension as proposed in [7] and Figure 19 gives these numbers after the removal of type I and type II redundancies followed by mask extension. We do not report the results of compaction using the enhancements to Liu's [7] compaction methods proposed in [8, 9], because, as noted in Section 4, these enhancements do not guarantee compacted prefix sets equivalent to the input prefix set. For each of our data sets, the method of [7] achieves less compaction than what is proposed for PETCAM. The bar chart in Figure 20 shows the relative efficiency of the three schemes with respect to reducing the number of prefixes in a router table.

DataSet	initial # of prefixes	# after type I	Reduction (%)	# after mask extension	Total reduction (%)
AS1221	281516	210582	25.20	146101	48.10
AS4637	210119	92099	56.17	40374	80.79
AS6447	275509	231193	16.09	162575	40.99
AS65000	259026	152267	41.22	80441	68.94
rrc00	266185	173030	35.0	105534	60.35

Figure 18: Number of router table prefixes in [7]

DataSet	initial # of prefixes	# after type I and II	Reduction (%)	# after mask extension	Total reduction (%)
AS1221	281516	209553	25.56	145467	48.33
AS4637	210119	92066	56.18	40386	80.78
AS6447	275509	227989	17.25	159909	41.96
AS65000	259026	151590	41.48	80076	69.09
rrc00	266185	171754	35.48	104827	60.62

Figure 19: Number of router table prefixes when type II redundancies are eliminated

The input for mask extension is comprised of sets of same hop prefixes that have the same length. Logic minimization is performed on each of these sets. The time for logic minimization is substantial (see Section 6.4) and critically dependent on the size of the input set. Figure 21 gives the maximum size of a set input to Espresso.

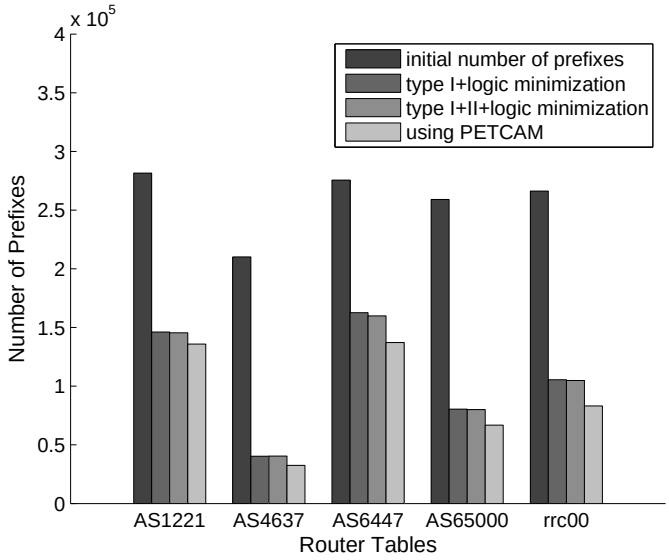


Figure 20: Relative efficiency for table compaction

Table	Original	PETCAM	Liu [7]
as1221	45285	13771	25953
as4637	111921	13188	40004
as6447	18808	7818	15110
as65000	94297	14693	45602
rrc00	62744	11669	34129

Figure 21: Maximum number of prefixes, having the same length and next hop prior to logic minimization/mask extension

6.2 Comparison With EaseCam

Although the modifications to Liu's [7] compaction strategy suggested in [8, 9] are faulty, the two-level EaseCam architecture, which is a specialization of the bit-selection architecture proposed by Zane et al. [19], may be employed in conjunction with any prefix set to reduce TCAM power as well as total TCAM memory. In EaseCam, each TCAM word is 24 bits as the first 8 bits of each prefix are used in the level-1 index to the TCAM. Prefixes shorter than 8 bits are handled in a separate bucket and are ignored in our evaluation of EaseCam. Figure 22 gives the number of TCAM bits required by EaseCam to store each of our sample router tables following compaction using the strategies of [7], [7] together with type II redundancy removal, and PETCAM.

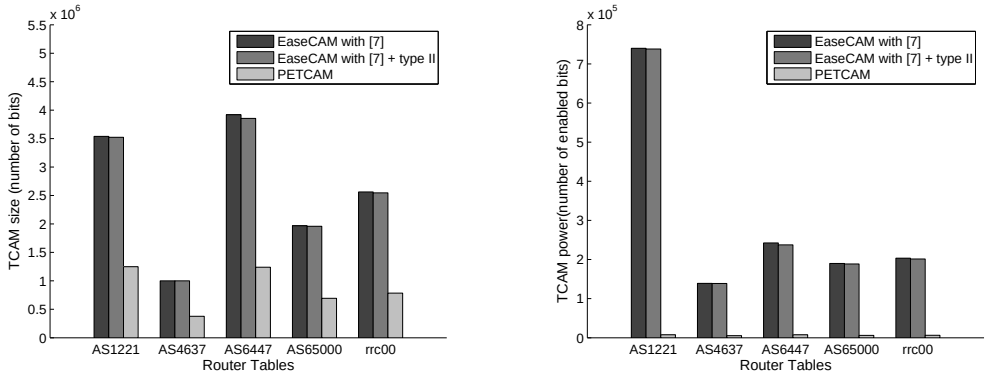
To generate the numbers for PETCAM, we employ steps 1 and 2 of the PETCAM scheme and store the resulting generalized prefixes in a 2-level TCAM system using the M-12Wb layout of [3]. We set the word-size of the second level TCAM (also known as data-DTCAM or DTCAM) to 32 bits for IPv4 prefixes. For the first level TCAM (index-TCAM or ITCAM), we need a word size of 24 bits based on (1) the bit allocation scheme to suffix nodes in Figure 7, (2) suffix node size of 144 bits and (3) our choice of DTCAM bucket size of 128 prefixes for the experiments. For example, with the given bit allocation and the suffix node size, the minimum height of

a carved subtree (carving done using the algorithm in Figure 15) is 2, corresponding to the case when all the 7 nodes of the subtree store a prefix. Thus prefixes stored in a DTCAM are of length 29 or less. Similarly, with a DTCAM bucket of size 128 prefixes, the carved subtree is of height at least 6 ($\log_2(128) - 1$, assuming a full binary tree with each node storing a prefix). So, the length of a prefix in ITCAM is at most 29-7 = 22, and in fact some further reduction is possible if we consider the wide SRAM being used with the ITCAM. However, for ease of configuration, we choose 32 bits for DTCAM and 24 bits for ITCAM word size. It is easy to see that this configuration can support DTCAM buckets of size >= 32 entries when the SRAM word size is 144 bits. When the DTCAM bucket size is < 32, a larger word size for the ITCAM is required.

Figure 22 also gives the size, $maxP$, of the largest partition that is activated in a lookup. Recall that the power needed for a lookup is proportional to the size of the activated partition. On our data sets, PETCAM requires less than half the TCAM memory required by EaseCam and the power requirement of EaseCam is between 26 and 97 times that of PETCAM. Figure 23 presents a bar chart for the comparitive space and power consumption by EaseCAM and PETCAM.

DataSet	EaseCam with [7]		EaseCam with [7]+typeII		PETCAM	
	#bits	$maxP$	#bits	$maxP$	#bits	$maxP$
AS1221	3538608	739968	3523656	738120	1248640	7552
AS4637	1000080	138936	1000080	138912	378056	5320
AS6447	3920112	242400	3854808	237360	1239608	7624
AS65000	1968408	189888	1959000	188736	694240	6112
rrc00	2563176	203448	2545920	201384	784592	6352

Figure 22: Total number of bits and maximum partition size using EaseCam



(a) TCAM size

(b) TCAM power

Figure 23: Comparison of TCAM space and power requirement between EaseCAM and PETCAM

6.3 Comparison With [3]

Figure 24 gives the $maxP$ values for the power-reduction architecture of Lu and Sahni [3] when applied to the original prefix set as is done in [3] and when applied to a compacted prefix set. In Figure 24, we use the 1-12Wc scheme of [3], which is recommended in [3] for power optimization. The size of a DTCAM bucket is set to 128 prefixes. In Figures 24 and 25, all columns use the architecture of [3]. The column labeled No Compaction [3] uses the original prefix set, that labeled [7] uses the compacted prefix set resulting from type I redundancy removal followed by logic minimization as is done in [7], the next column applies types I and II redundancy removal before logic minimization, and the column labeled PETCAM uses the 1-12Wc scheme to store the set of generalized prefixes obtained after applying steps 1 and 2 of the PETCAM scheme to the initial prefix set. As can be seen, PETCAM provides power reduction relative to the scheme of [3]. This reduction ranges from 18% to 25%.

DataSet	No Compaction [3]	[7]	[7]+type II	PETCAM
AS1221	200	180	180	164
AS4637	183	152	152	139
AS6447	196	185	185	164
AS65000	198	171	170	148
rrc00	198	172	172	152

Figure 24: Maximum partition size using 1-12Wc of [3]

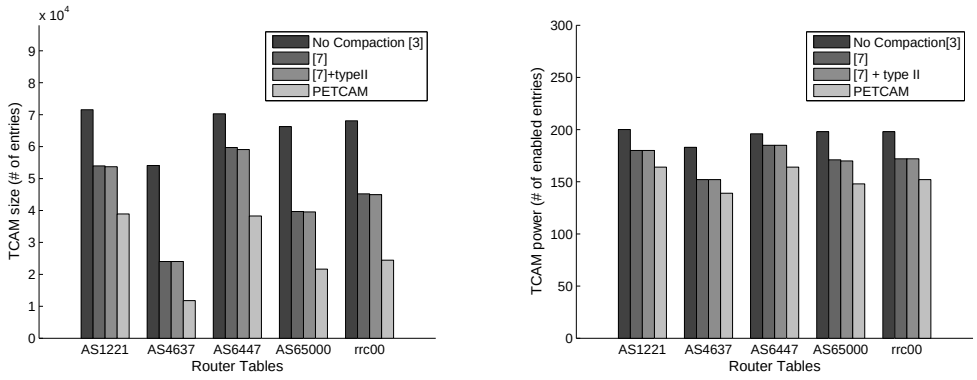
Figure 25 gives the total TCAM memory needed by the M-12Wb scheme of [3], which is the scheme recommended in [3] for TCAM memory optimization. The numbers in the column labeled PETCAM are obtained by applying the steps 1 and 2 of the PETCAM scheme to reduce the prefix set and then using the step 3 to map the resulting generalized prefixes to a 2-level TCAM system. We use the carving heuristic in Figure 15 to create the suffix nodes and then use the M1-2Wb layout of [3] to fill the first and second level TCAMs. The size of a DTCAM bucket is set to 128 prefixes. PETCAM requires between 22% and 54% as much TCAM memory as required by the architecture of [3] beginning with the original prefix set. Figure 26 shows the data of Figures 24 and 25 as bar charts.

DataSet	No Compaction [3]	[7]	[7]+type II	PETCAM
AS1221	71564	53964	53705	38913
AS4637	54076	24027	24027	11782
AS6447	70271	59728	59089	38273
AS65000	66285	39691	39560	21636
rrc00	68084	45216	44964	24449

Figure 25: Total TCAM memory using M1-2Wb of [3]

6.4 PETCAMLite

Since Step 2 (mask extension) of the compaction process for PETCAM is quite time consuming, we investigate a light version, PETCAMLite, of PETCAM in which Step 2 is omitted. Our experiments indicate that PETCAMLite



(a) TCAM size

(b) TCAM power

Figure 26: Comparison of TCAM space and power requirement

requires 0% to 6% more TCAM power and 0.5% to 2% more TCAM memory than required by PETCAM. So, if Step 2 takes more computational resource than we wish to invest, we may use PETCAMLite and gain almost the same power and memory benefits as provided by PETCAM. Figure 27 gives the CPU time on a Sun4u Sparc SunOS 5.8 machine for executing steps 1 and 2. So, if a Sun4u Sparc is used as the rebuild engine, the interval between successive rebuilds of the TCAM system will need to be at least 700 seconds for PETCAM but only about 6 seconds for PETCAMLite.

DataSet	Time for Step 1 (seconds)	Time for Step 2 (seconds)
AS1221	5.38	642.83
AS4637	3.7	296.62
AS6447	5.14	347.25
AS65000	4.57	600.55
rrc00	4.78	407.05

Figure 27: Execution time

7 Conclusion

We have pointed out some of the shortcomings of the power reduction methods for TCAM lookup tables proposed in [7, 8, 9]. By starting with an optimal prefix set for the given router table prefix set, we can achieve much better power reduction and TCAM memory requirement than when we use the compaction schemes suggested in [7, 8, 9]. This is true regardless of whether we use the EaseCam [8, 9] architecture or the architecture of [3]. For EaseCam, worst case power is reduced between 96% and 98% while TCAM memory is reduced between 62% and 69%. The power and memory reduction relative to the architecture of [3] is 16% to 25% and 45% to 78%. We have proposed two memory and power efficient TCAM lookup systems – PETCAM and PETCAMLite. While PETCAM has slightly better memory and power characteristics than does PETCAMLite, the rebuild time for PETCAM is 2

orders of magnitude larger than that for PETCAMLite. PETCAMLite supports acceptable rebuild times using modest computational resources. On our data sets, the power and memory penalty using PETCAMLite are at most 6% and at most 2%, respectively.

References

[1] M. Akhbarizadeh, M. Nourani, R. Panigrahy and S. Sharma, A TCAM-based parallel architecture for high-speed packet forwarding, *IEEE Trans. on Computers*, 56, 1, 2007, 58-2007.

[2] H. Lu, Improved Trie Partitioning for Cooler TCAMs, *ACST*, 2004.

[3] W. Lu and S. Sahni, Low Power TCAMs For Very Large Forwarding Tables, *Proceedings of INFOCOM*, 2008.

[4] W. Lu and S. Sahni, Succinct representation of static packet classifiers, *International Conference on Computer Networking*, 2007.

[5] *http://bgp.potaroo.net*, 2007.

[6] *http://www.ripe.net/projects/ris/rawdata.html*, 2008.

[7] H. Liu, Routing Table Compaction in Ternary-CAM, *IEEE Micro*, 22, 3, 2002.

[8] V.C. Ravikumar, R. N. Mahapatra, and L. N. Bhuyan, EaseCAM: An Energy And Storage Efficient TCAM-Based Router Architecture for IP Lookup, *IEEE Transactions on Computers*, 54, 5, May 2005, 521-533.

[9] V.C. Ravikumar, R. N. Mahapatra, and L. N. Bhuyan, TCAM architecture for IP lookup using prefix properties, *IEEE Micro*, 24, 2, March 2004, 60-69.

[10] R. Daves, C. King, S. Venkatachary, and B.Zill, Constructing Optimal IP Routing Tables, *Proceedings of INFOCOM*, 1999.

[11] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, Survey and taxonomy of IP address lookup algorithms, *IEEE Network*, 2001, 8-23.

[12] 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*, 14, 3, 2003, 337-358.

[13] C. A. Zukowski, and S. Wang, Use of Selective Precharge for Low-Power Content-Addressable Memories, *IEEE International Symposium on Circuits and Systems*, 1997.

[14] N. Mohan, and M. Sachdev, Low Power Dual Matchline Ternary Content Addressable Memory, *IEEE International Symposium on Circuits and Systems*, 2004.

[15] H. Miyatake, M. Tanaka, and Y.Mori, A design for high-speed low-power CMOS fully parallel content addressable memory macros, *IEEE Journal of Solid State Circuits*, 36, 6, June 2001, 956-968.

[16] C.-S. Lin, J.-C. Chang, and B.-D Liu, A low-power pre-computation based fully parallel content addressable memory, *IEEE Journal of Solid State Circuits*, 38, 4, April 2003, 654-662.

[17] Z. Wang, H. Che, M. Kumar, and S.K. Das, CoPTUA: Consistent Policy Table Update Algorithm for TCAM without Locking, *IEEE Transactions on Computers*, 53, 12, December 2004, 1602-1614.

[18] M. Wang, S. Deering, T. Hain, and L. Dunn, Non-random Generator for IPv6 Tables, *12th Annual IEEE Symposium on High Performance Interconnects*, 2004.

[19] F. Zane, G. Narlikar and A. Basu, CoolCAMs: Power-Efficient TCAMs for Forwarding Engines, *INFOCOM*, 2003.