EaseCAM: An Energy and Storage Efficient TCAM-Based Router Architecture for IP Lookup

V.C. Ravikumar, Rabi N. Mahapatra, *Senior Member*, *IEEE*, and
Laxmi Narayan Bhuyan, *Fellow*, *IEEE*

Abstract—Ternary Content Addressable Memories (TCAMs) have been emerging as a popular device in designing routers for packet forwarding and classifications. Despite their premise on high-throughput, large TCAM arrays are prohibitive due to their excessive power consumption and lack of scalable design schemes. This paper presents a TCAM-based router architecture that is energy and storage efficient. We introduce prefix aggregation and expansion techniques to compact the effective TCAM size in a router. Pipelined and paging schemes are employed in the architecture to activate a limited number of entries in the TCAM array during an IP lookup. The new architecture provides low power, fast incremental updating, and fast table look-up. Heuristic algorithms for page filling, fast prefix update, and memory management are also provided. Results have been illustrated with two large routers (`bbnplanet` and `attcanada`) to demonstrate the effectiveness of our approach.

Index Terms—Router, TCAMs, IP lookup, partition, compaction, page table.

1 Introduction

Internet protocol (IP) lookup forms a bottleneck in packet forwarding in modern IP routers because the lookup speed is unable to catch up with the increase in link bandwidth. Ternary Content Addressable Memories (TCAMs) have been emerging as viable devices for designing high throughput forwarding engines on routers. They store don't care states in addition to 0s and 1s and search the data (IP address) in parallel by providing a comparator in each cell. This property makes TCAMs particularly attractive for packet forwarding and classifications, where variable length prefix matching is necessary in every cycle. Despite these advantages, the use of TCAM arrays is prohibitive due to large power consumptions and lack of scalable design schemes. The power consumption in a TCAM is proportional to its size due to power drain at each cell.

The high density TCAMs, available in the market today, consume power up to 12-15W/chip when the entire memory is enabled. In order to support the IP prefixes that are increasing at a significant rate, vendors use four to eight such TCAM chips for IP lookup operation. More chips would also be required to handle filtering and packet classification, if implemented by TCAMs. The power consumption resulting in using a large number of chips not only increases the cooling costs, but also limits the router design to fewer ports. The power consumption increases linearly with the increase in the number of entries in a TCAM. The research effort to reduce power consumption in TCAM can be divided into two categories. The first category attempts to partition the entire routing table with some suitable hashing technique, as discussed in [8], [10]. Only one partition is energized at a time. This approach reduces the power consumption by $\frac{n \times p_{\max}}{N}$, where "$p_{\max}$" is the size of worst case partition size generated by the hashing scheme, "n" is the number of partitions enabled, and "N" is the size of the routing table. The second approach attempts to compact the routing table entries by removing redundancy using logic minimization and pruning techniques, as discussed in [9]. This technique reduces the power consumption by 30-40 percent depending on the performance of algorithm used for logic minimization. However, we show that this technique [9] takes excessive time for updating because this is based on the Espresso-II minimization algorithm, whose complexity increases exponentially with the size of the original routing table [4]. The Espresso-II is a two-level algorithm that employs operations like expand, reduce, irredundant, and essentials for logic minimization. The algorithm is unsuitable for handling frequent (about 30-60 seconds interval) routing table updates even if initial minimization time is tolerable.

The main motivation of this paper is to devise a TCAM router architecture that consumes low power and is suitable for the incremental updating needed in modern IP routers. Additionally, it is intended to minimize the effective memory size and look-up delay. We propose a pipelined architecture that can achieve the above goals through further compaction of the active region of the routing table in TCAM architecture. In this paper, we introduce the idea of *prefix aggregation* and *prefix expansion* for TCAM which can reduce the number of entries for comparison in a TCAM. Together with the prefix aggregation and overlapping properties, we select a limited number of pages in the proposed architecture during IP look-up instead of energizing the entire memory module. The present TCAM vendors provide mechanisms to enable a chunk of TCAM,

- *V.C. Ravikumar and R.N. Mahapatra are with the Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. E-mail: {vcr, rabi}@cs.tamu.edu.*
- *L.N. Bhuyan is with the Department of Computer Science, University of California, Riverside, CA 92521. E-mail: bhuyan@cs.ucr.edu.*

Manuscript received 16 Jan. 2004; revised 2 Aug. 2004; accepted 16 Sept. 2004; published online 16 Mar. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0020-0104.

much smaller compared to the entire TCAM. We exploit this technology to achieve an upper bound on the power consumption in the TCAM-based router. Based on a statistical analysis of the current IP routing tables, we present a 2-level paged TCAM architecture. Detailed analysis and design trade offs are presented by varying the subprefix length. We apply the logic minimization technique used in [9] to further compact each of the partition, thus reducing the worst-case partition bound further by 30-40 percent. We also propose a page filling heuristic to improve memory utilization due to paging. The concept of bucket-TCAM has been used to isolate storing of rarely incoming IP prefixes and to find optimal page size in the proposed architecture. An efficient memory management scheme is presented for updating the routing entries. Finally, we derive empirical equations for memory requirement and power consumption in the proposed architecture.

Case studies were made using the statistics from `bbnplanet` and `attcanada` routers to demonstrate the effectiveness of the proposed TCAM architecture. It is shown that less than 1 percent of power is consumed when compared to energizing the full-size TCAM implementation. Considerable saving is also noticed in memory size and look-up delay.

The paper makes the following significant contributions:

- It introduces prefix-aggregating technique to substantially reduce TCAM size.
- It presents a two-level pipelined architecture based on which selected pages and modules can be activated during a table lookup process.
- Efficient paging and memory management techniques are derived to improve TCAM utilization and update operations.
- Through analysis and simulation, it demonstrates a substantial reduction in power and memory size when the proposed architecture is used.

The rest of the paper is organized as follows: Section 2 introduces some definitions and presents routing table compaction and partitioning techniques based on some of the prefix properties. Section 3 discusses the proposed architecture for the forwarding engine and tackles the problem of optimizing power and memory. The routing table update mechanism and memory management have been addressed in Section 4. Section 5 illustrates the results and case studies based on the statistics from two real routing tables. A brief summary on related work is given in Section 6. In Section 7, we conclude this work.

2 Routing Table Compaction and Partitioning

Routing table compaction is achieved by exploiting several properties, exhibited by IP prefixes. The compaction approach, used in [9], exploits *pruning* to remove redundant overlapping entries and *mask extension* to combine entries using logic minimization. *Pruning* achieves compaction by storing only one of the many *overlapping prefixes* that have the same next-hop. Two prefixes are overlapping if one is an enclosure of the other. Let P_i denote the prefix P_i and $|P_i|$ denotes the length of prefix P_i, then $P_i \in P$ is called *enclosure* of P_j if $P_j \in P, j \neq i$ and $|P_i| < |P_j|$ such that P_i is a subprefix of P_j. If P_i and P_j have the same next hop, then they can be represented by P_i. Thus, a set of overlapping prefixes $\{P_1, P_2, P_3, \ldots P_n\}$ such that $|P_1| < |P_2| < |P_3| \ldots < |P_n|$, having the same next hop, can be replaced with a single entry P_1. If an update deletes the entry P_1, the set of overlapping prefixes $\{P_1, P_2, \ldots P_n\}$ should be represented by the entry P_2. When an update adds a new entry P_i such that $|P_i| < |P_1| < |P_2| \ldots < |P_n|$, then the existing entry P_1 is replaced with P_i. However, if the new entry P_i arrives such that $|P_i| > |P_1|$, then no changes are made to the routing table except for the updating of set of overlapping prefixes.

The mask extension property combines two prefixes of the form $s_1 0 s_2$ and $s_1 1 s_2$ to $s_1 x s_2$ if they have the same next hop. The promotion of "0" and "1" to a don't care symbol "x" allows us to keep one entry while maintaining the search semantics. A set of prefixes having the same nexthop is reduced to a minimal set by applying mask extension repeatedly. This procedure is similar to *logic minimization* and, hence, standard tools available for logic minimization can be used to implement mask extension. In order to perform compaction, the routing table is first partitioned into subtables based on the nexthop. Pruning is applied to each of these subtables to remove overlapping entries and then each of the resulting subtables is further partitioned based on the prefix lengths. A logic minimization algorithm, Espresso-II, is applied to each of these subtables to achieve mask extension. This partitioning scheme, however, concentrates a large number of entries in a few partitions. For example, the largest partition in the `attcanada` routing table contains more than 15,000 routing entries. As mentioned earlier, the Espresso-II algorithm has exponential behavior and, therefore, unsuitable for minimization or incremental update of large-size subtables such as above.

Another traditional way to obtain low power consumption is to partition the routing table hierarchically starting with the most significant bits (msb) [8]. Based on w_1 msbs, the routing table can be partitioned to 2^{w_1} smaller segments. Taking the next w_2 msbs, these smaller segments can be further subdivided into 2^{w_2} parts and so on until a desirable size of the TCAM module is achieved. When a packet arrives, one of these TCAMs is energized depending on the msb of the IP address. While this is a simple approach to reducing power, the scheme is not realizable in practice because it requires too many TCAMs with very few entries, giving rise to poor utilization. We propose to preserve this simple partitioning scheme by adopting a two-level architecture based on a certain value of w_1 and supplement it with a prefix aggregation scheme to improve utilization. In Section 3, we evaluate the performance based on varying w_1 and adopt $w_1 = 8$ for implementation.

The prefix aggregation technique reduces the minimization and update times at an acceptable compaction loss. However, this compaction loss can be compensated for, by using another property called *prefix expansion*. The prefix expansion increases the minimization efficiency without adding significant execution overhead. We describe the prefix aggregation and expansion in the following subsections.

TABLE 1
Sample Routing Table Trace—bbnplanet Router

IP Address	nexthop	IP Address	nexthop	IP Address	nexthop
65.117.15.240/29	205.171.0.61	65.117.15.176/29	205.171.0.61	65.117.15.112/29	205.171.0.61
65.117.15.216/29	205.171.0.61	65.117.15.160/29	205.171.0.61	65.117.15.96/29	205.171.0.61
65.117.15.208/29	205.171.0.61	65.117.15.152/29	205.171.0.61	65.117.15.72/29	205.171.0.61
65.117.15.200/29	205.171.0.61	65.117.15.136/29	205.171.0.61	65.117.15.64/29	205.171.0.61
65.117.15.192/29	205.171.0.61	65.117.15.128/29	205.171.0.61	65.117.15.56/29	205.171.0.61
65.117.15.184/29	205.171.0.61	65.117.15.120/29	205.171.0.61	65.117.15.40/29	205.171.0.61

TABLE 2
Evaluation of Aggregation Performance on Compaction

Router	Total Prefixes	Maximum Compaction	Prefix Aggregation based Compaction
aads	33740	18440	18608
attcanada	112412	54476	57837
bbnplanet	124538	69646	71500
maewest	29585	16305	16462
pacbell	22165	11222	11388
paix	13914	8863	8893

2.1 Prefix Aggregation

Once, a two-level scheme is adopted, the large segments from two-level partitioning are further subdivided using prefix aggregation. This partitioning scheme generates a much more uniform and bounded partitions.

Prefix Aggregation is defined as a process by which the prefixes $P_\ell = p_1p_2p_3...p_\ell$ in a routing table are partitioned based on the value of a maximum subprefix such that

$$m = \begin{cases} \ell & when \quad \ell < 8 \\ \lfloor \frac{\ell}{8} \rfloor & \times \quad 8 \quad otherwise, \end{cases}$$

where $p_i \in [0, 1]$. The length of the maximum subprefix is chosen as a multiple of 8 for simplicity, which also corresponds to a value of $w_1 = 8$, chosen to implement two-level partitioning. Since all the prefixes present in a partition have the same value of the *subprefix as defined above*, we term it the *Largest Common Subprefix (LCS)*. For example, LCS for the prefixes 129.66.30.0/23 and 129.66.20.0/22 is 129.66.0.0/16 according to the definition given above. Table 1 shows portions of the routing table dump taken from the bbnplanet router. These prefixes have their length 29 and the LCS 65.117.15.0/24 and, therefore, belong to the same partition after prefix aggregation.

We have analyzed traces from six other routers, listed in Table 2. Statistical observations show that if the prefixes are partitioned according to the proposed partitioning scheme, they correspond to the same next-hop with a very few exceptions. We applied the Espresso-II algorithm to individual partitions generated by the above prefix aggregation technique and compared results to the best case when the algorithm is applied to the entire routing table for logic minimization. It may be noticed from Table 2 that prefix aggregation-based compaction results in nearly maximum prefix compaction. Since aggregated partitions are much smaller compare to nonaggregated partitions, it saves considerable compaction time by limiting the input data size to the Espresso algorithm. As a result, the proposed technique is more suitable to incremental (online) updating as opposed to the technique in [9].

The partitioning scheme based on prefix aggregation yields a fairly uniform and bounded distribution. The worst-case bound on the number of entries in the largest partition is established in following lemma.

Lemma 1. *The maximum number of prefixes in an LCS-based partition $LCS(P)$ is 2^8.*

Proof. Let $P_1 = p_1p_2p_3 \dots p_{\ell_1}$ and $P_2 = p_1p_2p_3 \dots p_{\ell_2}$ be two arbitrary prefixes such that $\ell_1 \geq 8$ and $\ell_2 \geq 8$. If the prefixes P_1 and P_2 have the same $LCS = p_1p_2p_3 \dots p_m$, we can conclude from *aggregation definition* that $m = \lfloor \frac{\ell_1}{8} \rfloor \times 8$ for P_1 and $m = \lfloor \frac{\ell_2}{8} \rfloor \times 8$ for P_2 or we can write $\lfloor \frac{\ell_1}{8} \rfloor \times 8 = \lfloor \frac{\ell_2}{8} \rfloor \times 8$, which yields $|\ell_1 - \ell_2| < 8$. Thus, the maximum prefix length difference among the prefixes having the same LCS is bounded. Therefore, the maximum number of prefixes having the same LCS is given by $2^8 = 256$. □

2.2 Prefix Expansion

Espresso-II gives more compaction performance if all the inputs (set of prefixes) given to the algorithm for minimization are of the same length. The prefix expansion technique reduces a set of arbitrary length prefixes to a predefined set of lengths. The details on prefix expansion for IP lookup have been discussed in [6]. We adopt this technique here to further compact the routing table. However, in our case, the prefix expansion takes place at octet steps. The expansion of a prefix is described as a set of strings $E(P_\ell) = \{P_\ell \bullet i | 0 \leq i < 2^{m+8-\ell} \, and \quad i \in [0, 1]^{m+8-\ell}\}$. The operator • denotes the concatenation operation and m

is as given in prefix aggregation. Here, an arbitrary length prefix P_ℓ is expanded to its nearest octet position, which gives rise to $2^{m+8-\ell}$ entries and is formed by concatenating the binary representation of P_ℓ. The binary representation of these numbers is between 0 and $2^{m+8-\ell}-1$. For example, the prefix 129.66.30.0/23 should be expanded into prefixes 129.66.30.0/24 and 129.66.31.0/24. Please note that the prefix expansion adds $2^{m+8-\ell}-1$ entries to the existing routing table. However, Lemma 1 places an upper bound on the size of any partition to be 256, therefore the minimization time remains bounded. In fact, additional benefits can be achieved by ignoring the first "m" bits as they remain constant within any partition. This reduces the size of each input given to Espresso-II, thereby reducing the execution time further.

2.3 Prefix Overlapping

As defined earlier, two prefixes are called overlapping if one is an enclosure of the other. The *pruning* procedure suggested in [9] removes the longer of the two overlapping IP prefixes corresponding to the same next hop to boost compaction performance without affecting search semantics. However, if the overlapping prefixes correspond to different next hops, they are not pruned out and search for a particular IP address will try matching against all of the corresponding overlapping prefixes. According to an observation made in a recent study [13], most of the IP prefixes have their lengths between 8 and 24. The study also suggests rapid growth of prefixes having lengths between 25 and 32. Noting that there are no prefixes which have length below 8, we can observe that *the maximum number of overlapping prefixes* is $32-7=25$. If all the overlapping prefixes are not confined to a few partitions, the benefit of our partitioning scheme is defeated. This is because all the partitions with overlapping prefixes will have to be searched. In the worst case, we may have to search up to 25 segments, resulting in high power consumption. Lemma 2 presents an upper bound on the number of prefix aggregation generated partitions containing all the overlapping prefixes for a given IP address. Lemma 3 presents an upper bound on the total number of entries to be searched for a given IP address in the LCS-based partitioning scheme.

Lemma 2. *The maximum number of partitions generated by LCS aggregation and containing all the overlapping prefixes for a given IP address is* $\lfloor \frac{25}{8} \rfloor$.

Proof. From Definition 1, it follows that all the overlapping IP prefixes whose length "ℓ" satisfies $\mu \times 8 \leq \ell \leq (\mu+1) \times 8$ for some integer μ, have the same LCS and, hence, belong to same partition. Therefore, the maximum number of partitions occupied can be obtained by dividing the maximum number of overlapping prefixes possible by 8. From earlier discussion, we know that the maximum number of possible overlapping prefixes for a given IP address is 25. Therefore, the maximum number of partitions can be given by $\lfloor \frac{25}{8} \rfloor$. □

Lemma 3. *The maximum number of entries to be searched in the LCS-based partitioning scheme for a given IP address is* 3×256.

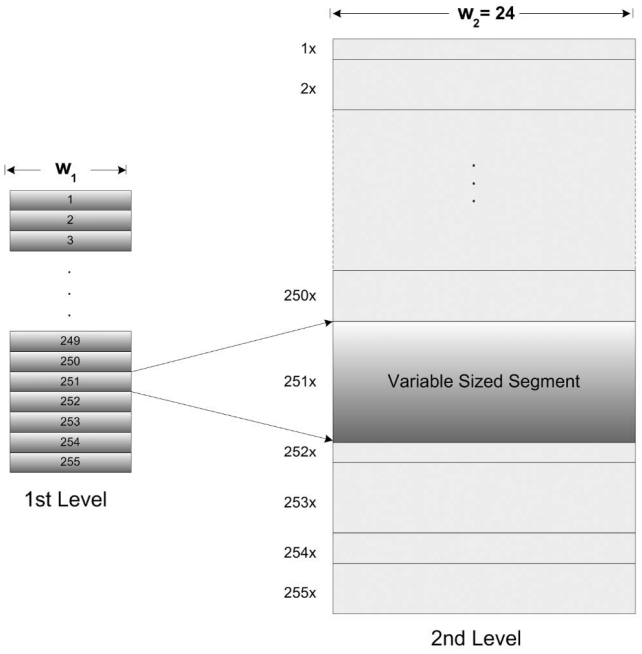


Fig. 1. Segmented architecture for IP lookup using TCAM.

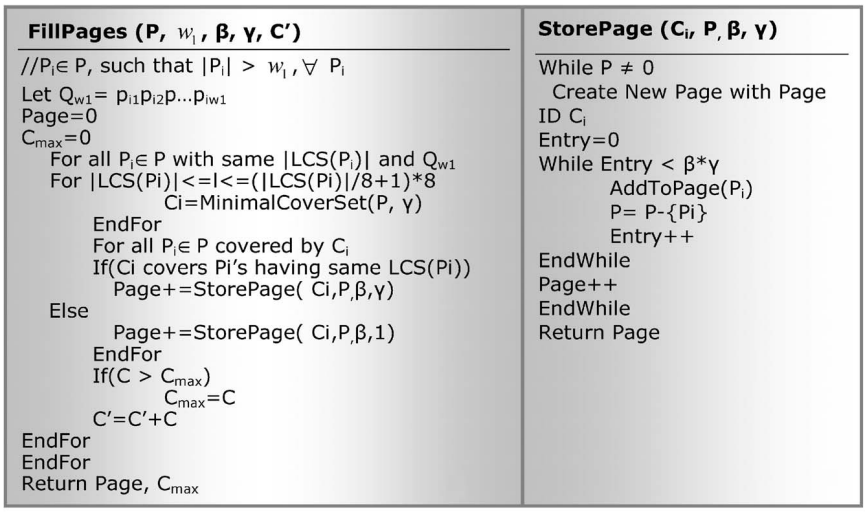
Proof. For as LCS-based partitioning scheme, the number of partitions to be searched for a given IP address is $\lfloor \frac{25}{8} \rfloor$. Since the worst-case partition size is 2^8, as given in Lemma 1, it follows that the maximum number of entries to be searched equals $\lfloor \frac{25}{8} \rfloor \times 2^8 = 3 \times 256$. □

The proposed approach combines routing table compaction and partitioning mentioned above to implement an efficient TCAM-based IP lookup architecture. Here, the routing table is first partitioned into subtables according to the w_1 most significant bits of the prefixes. Thus, we consider only those prefixes which have their length greater than w_1. Each of these subtables is further partitioned according to nexthop. A pruning procedure is applied to each of these subtables to remove redundant overlapping entries. The entries in each of the resulting subtables are permuted to aggregate the prefixes having the same LCS together. The permuted subtables are then expanded using the prefix expansion technique and subjected to minimization using the Espresso-II algorithm. The minimized subtables are then distributed to a number of fixed-size TCAM pages. The next section presents a detailed two-level pipelined IP lookup architecture based on the above techniques.

3 Design of Router Architecture

3.1 TCAM Lookup Architecture

The proposed two-level TCAM-based IP lookup engine is shown in Fig. 1. The first level contains w_1-bit subprefixes which are compared against the w_1 most significant bits of incoming IP addresses. If there is a match in the first level, it enables the corresponding region in the second level. This region is called segment in Fig. 1 and is indexed by the entries in the first level. The architecture exploits the flexibility of modern TCAMs to selectively enable a portion



FillPages (P, w_1, β, γ, C′)

```
//Pi∈ P, such that |Pi| > w1 ,∀ Pi
Let Qw1= pi1pi2p...piw1
Page=0
Cmax=0
   For all Pi∈ P with same |LCS(Pi)| and Qw1
   For |LCS(Pi)|<=I<=(|LCS(Pi)|/8+1)*8
            Ci=MinimalCoverSet(P, γ)
      EndFor
      For all Pi∈ P covered by Ci
      If(Ci covers Pi's having same LCS(Pi))
        Page+=StorePage( Ci,P,β,γ)
   Else
        Page+=StorePage( Ci,P,β,1)
      EndFor
      If(C > Cmax)
            Cmax=C
      C′=C′+C
EndFor
EndFor
Return Page, Cmax
```

(a)

StorePage (C_i, P, β, γ)

```
While P ≠ 0
  Create New Page with Page
ID Ci
Entry=0
While Entry < β*γ
      AddToPage(Pi)
      P= P-{Pi}
      Entry++
EndWhile
Page++
EndWhile
Return Page
```

(b)

Fig. 2. Algorithm 1. (a) Page Filling Heuristics. (b) Heuristics for storing prefixes into pages.

of TCAM to implement variable-sized segments resulting from proposed partitioning scheme. These variable-sized segments are implemented using fixed-size TCAM pages in level 2 after pruning, prefix expansion, and mask extensions using Espresso-II, as described in [9].

Due to the high variability of segment sizes, the implementation may cause significant underutilization of available memory. The following subsections discuss *page filling heuristics* to uniformly distribute the entries over all the pages and a method for determining *optimal page size*. We then *justify* the optimal value of 8 chosen for w_1 from measurements and statistical analysis of several routing traces and worst-case bound on the partition size. We also present an empirical measure of the total memory requirement in the proposed architecture. Finally, we describe the detailed TCAM architecture based on the concept developed throughout the section.

3.2 Page Filling Heuristics

The proposed partitioning technique generates uniform and bounded partitions by clustering smaller partitions together in order to increase memory utilization. Here, we present page filling heuristics that support the implementation of this clustering approach. Normally, the segments characterized by the set $\mathbb{S}_\psi = \{P_i | LCS(P_i) = \psi, P_i \in \mathbb{S}\}$ over the routing table can be indexed using ψ as segment identity. Please note that ψ represents the LCS value of a prefix and therefore can be treated as a fixed-length ternary string in $[0, 1, x]^W$, where "W" is the length of IP addresses. The ternary strings can be visualized as a set of points on an n-dimensional hypercube. A ternary string ψ_1 is said to *cover* another ternary string ψ_2 if the points represented by ψ_1 on the hypercube include all the points represented by ψ_2. For example, 10x represents a set of two points $\{(1,0,0),(1,0,1)\}$ on a three-dimensional hypercube, which is covered by 1xx representing the point set $\{(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}$ but not covered by 11x representing the point set $\{(1,1,0),(1,1,1)\}$. We also refer to these ternary strings as *cubes*. In order to be able to cluster the smaller segments together, we need a mechanism to cluster their segment identities. The proposed heuristics computes a cover of the set $\mathrm{A} = \{\psi_j | \mathbb{S}_{\psi_j} \subset \mathbb{S}\}$ to get

$$\mathrm{A}' = \left\{ \psi'_j \middle| \mathrm{cover}\left(\bigcup_{j=1} \psi_j\right) = \bigcup_{j=1} \psi'_j \right\},$$

using a logic minimization technique, such that their cardinalities satisfy $|\mathrm{A}'| \leq |\mathrm{A}|$. We designate ψ'_js which are much smaller in numbers to represent the *page identity*. Thus, page identity constructed in this way can index many segments in a single page.

The method is summarized in Algorithm 1, shown in Fig. 2, which computes page identities and fills the pages efficiently. The algorithm uses a *fill factor* γ, which represents the fraction of a page filled during initial configuration or reconfiguration in order to accommodate future updates. The fill factor can be statistically determined based on the static routing trace analysis to approximately support a specified number of future updates. The page size here is assumed to be "β" and its optimal determination is discussed later in this section. The page filling algorithm considers all prefixes with the same first level subprefix and LCS value for logic minimization. First, it tries to form coverings of maximum size ($\beta * \gamma$) and fills them into the same page. However, if the number of prefixes having the same LCS value is very small, it tries to combine prefixes have the same first level subprefix to allow maximum storage utilization. The `MinimalCoverSet` function finds the minimal number of cubes that are nonoverlapping and cover all the prefixes in the same page. The StorePage algorithm ensures that no page has more than $\beta * \gamma$ entries.

3.3 Bucket

The proposed page filling heuristics does not handle the prefixes whose length is smaller than w_1. We use a separate set of pages, referred to as *bucket*, to store these prefixes. The entries in the bucket will have to be searched only when there is a mismatch in the first level. The bucket size behavior is shown in Fig. 3 in terms of the fraction of entries

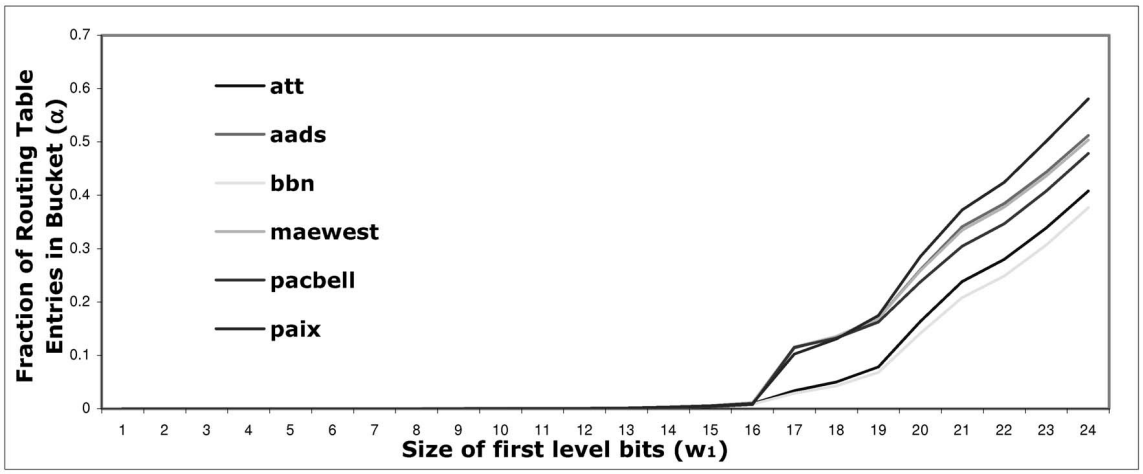


Fig. 3. Bucket size behavior with respect to size of bits used in the first level.

in the routing table termed "α" against the number of bits used in the first level w_1. The number of routing table entries present in the bucket is quite small for smaller values of w_1 and very unlikely to cause a bucket overflow with some suitable fill factor. This is also facilitated due to little growth trend toward prefixes of smaller size [13]. However, the bucket size increases almost linearly for larger values of w_1 and quickly becomes a controlling factor in worst-case power consumption.

We also utilize the bucket to store the new entries till reconfiguration, which cannot be stored in the designated pages due to page overflow. The number of such entries will be quite small with a suitably chosen page fill factor. The reconfiguration is triggered by memory management when page table overflow occurs.

3.4 Empirical Model on Memory Size

We now present empirical models to compute the total memory requirements and energy consumption for the proposed architecture. The minimum memory requirement for storing N, W-bits prefixes in the proposed architecture is:

$$M = \frac{\beta_{w_1} \times Page_{\max} \times (W - w_1)}{W} + Page_{\max} + \frac{Page_{\max}}{C_{\max}} + \frac{N \times \alpha}{\gamma_{bucket}}. \quad (1)$$

The first term in the given expression corresponds to memory required by the pages to store prefixes in the second level. The first w_1 bits are not stored as they are utilized only in the first level to select appropriate second level pages. The term β_{w_1} and $Page_{\max}$ refer to page size and maximum number of pages, respectively, for a given value of w_1. The second term corresponds to memory used to store $Page_{\max}$, W-bits page identities in the first level. The number of comparators used can be approximated to the number of page tables and is given by $C_{\max}$. The bucket size is represented by last term where $N \times \alpha$ represents the number of prefixes in the bucket during reconfiguration time and γ_{bucket}, the bucket fill factor. Please note that we take $W = 32$ for IPv4 prefixes.

3.5 Optimal Page Size

An optimal page size is essential for better memory utilization and, consequently, the power consumption. For example, if the page size were large, having separate pages for the smaller sized segments would result in wastage of space. Similarly, if the segment sizes are large, having small sized pages will increase the number of page IDs and, hence, the memory utilization.

We now present the heuristics to obtain the optimal page size $\beta_{optimal}$ for all possible w_1 and all possible values of page sizes, which will optimize the memory and, hence, the power consumption. The heuristics is described in Algorithm 2, shown in Fig. 4. The algorithm considers page sizes from 2 to 256 and computes the memory requirement using (1) for all values of w_1. The algorithm outputs the value of page size for which the memory requirement is minimal.

3.6 Deriving the value of "w_1"

We assess the performance of the proposed architecture in terms of power savings, compaction efficiency, and execution time for routing table minimization and updates. The performance of architecture is highly dependent on the value of "w_1" selected to implement the LCS partitioning. In order to determine the most appropriate value, we collected the routing table compaction and minimization time on

```
ForEach w1 from 8 to 31
//These are the possible values of w1
        ForEach j from 1 to 8
                //β ranges from 2 to 256, powers of 2
                β = 1<<j
                C'=0
                Page=FillPage(P,w1,β,γ,C')
                //Fill the page using heuristics
                Mem= Page* β*(32-w1)/32 + Page + Mc
                // Mc memory of comparators used
                If(Mem < Memw1)
                        //Obtain optimal β value
                        Memw1=Mem
                        Pagemax=Page
                        βw1= β
                EndIf
        EndFor
EndFor
```

Fig. 4. Algorithm 2. Optimal Page Size Determination.

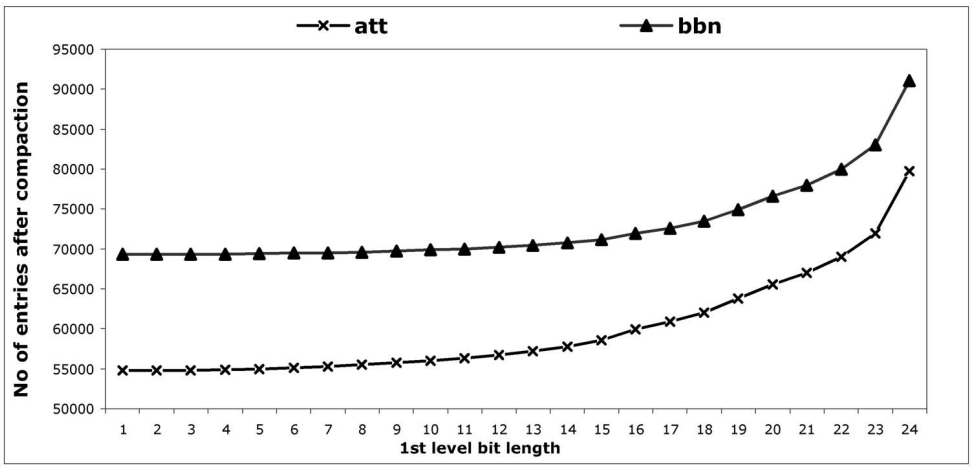


Fig. 5. Compaction performance with first level bit-length for routers with over 100,000 entries.

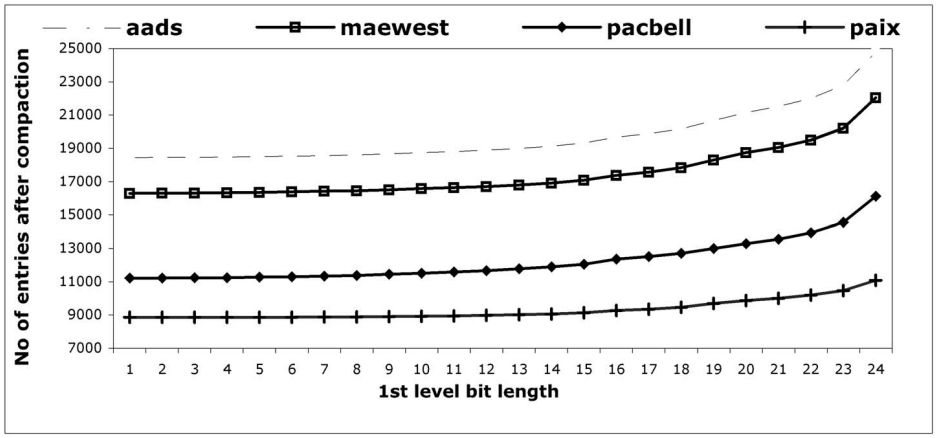


Fig. 6. Compaction performance with first level bit-length for routers up to 30,000 entries.

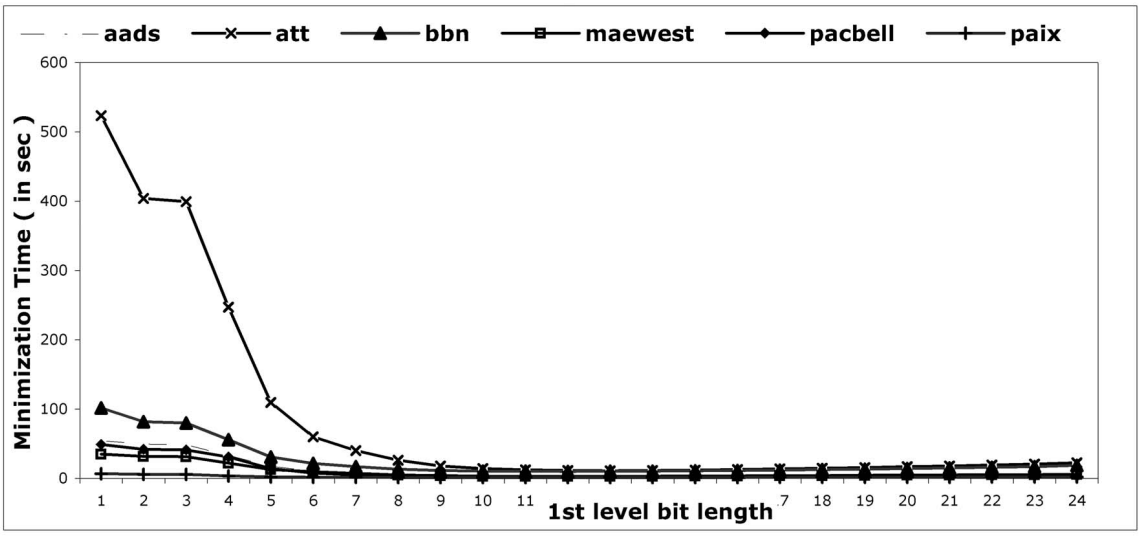


Fig. 7. Minimication time due to first level bit-length used in partitioning.

several router traces for different values of "w_1." The results have been summarized in Fig. 5, Fig. 6, and Fig. 7.

Fig. 5 shows the compaction performance on big routing table traces from `bbnplanet` and `attcanada` with respect to varying the value of "w_1." Fig. 6 also shows the compaction performance, but on smaller routing table traces from `aads`, `maewest`, `pacbell`, and `paix`. Please note that, for both of these categories, the compaction performance increases with decreasing value of "w_1." Hence, we would like to select "w_1" as lower as possible.

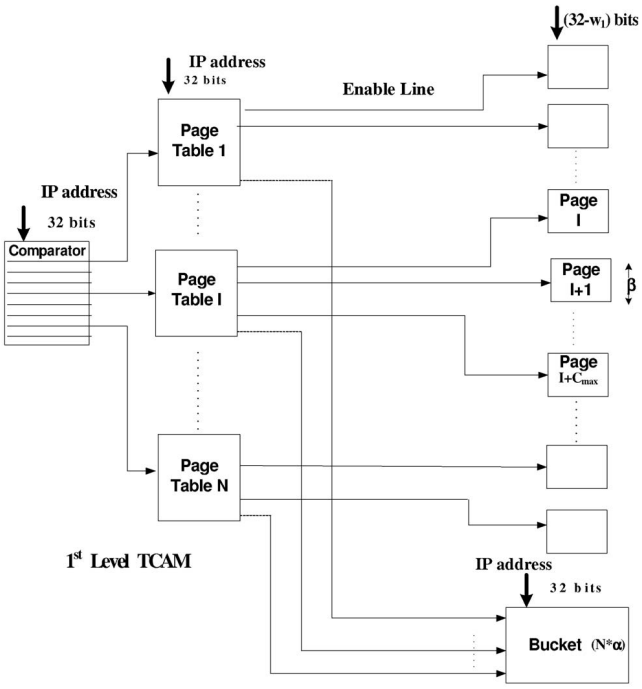


Fig. 8. The detailed TCAM architecture.

Fig. 7 shows the execution time for compacting the routing table with respect to "w_1." The execution time, unlike compaction, increases rapidly with decreasing value of "w_1." Due to frequent routing table updates with time interval between 30 and 60 seconds, we would like to keep the execution time less than 30 seconds. Fig. 7 suggests the value of $w_1 \geq 8$ in order to fulfill this requirement. We choose $w_1 = 8$, the lower end of the permissible values in order to get the maximum compaction. As we can notice from Fig. 5 and Fig. 6, the compaction doesn't improve very much for $w_1 < 8$. The following subsections present the architectural features based on the optimal value of $w_1 = 8$.

3.7 Determining Bound on Page Size

From Lemmas 1-3, the worst-case partition size, the maximum number of partitions containing overlapping prefixes, and maximum number of entries to be searched are 2^8, 3, and 3×256, respectively. The maximum size of the page table containing page identities can be computed by realizing that there can be a maximum of three overlapping *cubes* for a given IP address, if the *cubes* in each *cover* are made nonoverlapping. This ensures that the maximum number of active or enabled entries cannot exceed $256 * 3$. This will be true if all the prefixes in the *cube* C_i can be arranged in pages so that the total number of entries in all pages does not exceed 256.

3.8 Detailed TCAM Architecture

Fig. 8 shows the detailed architecture of the proposed approach. The first level lookup consists of comparators and page tables and the second level lookup consists of TCAMs in the form of actual pages and the bucket. The comparator primarily selects and enables page table with entries corresponding to the pages that contain the longest matching prefix. When an IP address is looked up using the forwarding engine, depending on the *cube* that covers the IP address, the comparator enables the appropriate page table entries. For example, the prefixes 128.1.5.0/24 and 128.1.4.0/24 can be represented by a single *cube* 128.1.2.0/23, which is stored in the page table. The 32-bit IP address is searched in the page table and, if there is a match, all the pages covered by the prefix are enabled. The least significant 24 bits of the IP address are then looked up in the enabled pages to find the longest prefix match. However, if there is no match in the page table, the 32-bit IP address is looked up in the bucket for a match.

The page tables store the page IDs associated with the pages. These page tables are 32-bit wide and implemented using TCAMs. The width of page tables is 32-bit because the maximum length of the *cube* can be 32 in the worst case. The minimum size of the page table is given by $C_{\max}$, as computed in Algorithm 1. The page tables have some empty entries to accommodate future updates. These entries are initialized with all zeros so that they do not match with incoming IP addresses.

The range comparator is designed (using TCAM) such that it would selectively enable the TCAM page table that contains the cubes covering the incoming IP address. From the router statistics, we know that the number of page tables required is few in numbers. Thus, the number of comparators required is small (i.e., `bbnplanet` statistics show that the number of comparators varied from 14 to 308 for different values of w_1). Hence, the overheads due to comparators for memory usage and power consumption can be considered negligible. It is important to note that the bucket, page tables, and comparators are all of 32-bit word length, while the pages in the TCAM have word length of $32 - w_1$. Thus, we cannot place them in the same TCAM chip modules. However, the total memory for the buckets, page tables, and comparators will be of very small size and they can be implemented using smaller sized TCAM chips of word length 32 bits.

4 Incremental Updates

Due to internet route instability, the routing table is subjected to continuous updates. The peak update rates can sometimes reach 2,000 updates per seconds in modern backbone routers, requiring fast and efficient incremental update mechanisms. The incremental update discussed in [9] takes significant execution time if the updates are made to partitions containing a large number of entries. The update algorithms presented here derive their efficiency from the bound placed on the partition size due to prefix aggregation.

4.1 Insertion

The insertion procedure for the proposed TCAM architecture is described in Algorithm 3, shown in Fig. 9. As explained earlier, the page table contains the cubes, which represent a cover of the prefixes present in a TCAM page. Therefore, when a prefix P_i is to be added, the page table is searched in order to find the page identities covering the given prefix.

Based on the page filling heuristics described earlier, we should insert the given prefix P_i into a target page indexed

```
Insert(Pi)
If( |Pi| ≤ wl)
InsertIntoBucket(Pi)
Else
If(∃ Ci that covers Pi)
      find Ci such that
      |LCS(PCover(Ci))|= |LCS(Pi)|
      // PCover(Ci) returns a prefix
      // covered by Ci
      P=Cover(Ci)
      P=P+{Pi}
      Q=Minimize(P)
      DeleteFromTCAM(Q' ⋂ P)
      InsertToTCAM(Q⋂P')
      UpdatePageTable(Ci)
Else
      InsertToBucket(Pi)
```

Fig. 9. Algorithm 3: Incremental Insertion.

by page identity C_i such that $LCS(P_i) = LCS(C_i)$. The target page is reminimized after inserting P_i and subsequently updated in TCAM. The reminimization may also affect the page identities referring to the target page. Therefore, all the page identities corresponding to the target page are recomputed and updated in the page table. The given prefix P_i is inserted into the bucket if it is not covered by any of the page identities.

4.2 Deletion

The deletion procedure for the proposed TCAM architecture is described in Algorithm 4, shown in Fig. 10. The deletion procedure is similar to the insertion procedure, but is simplified by the fact that the reminimization is done on the raw prefix data and not on the previously minimized set. In order to delete a prefix P_i , we first search for it in the bucket. If the prefix is found in the bucket, it is simply deleted. If the prefix is not found in the bucket, it is searched for in the page table to find the page identity C_i covering P_i. The prefix P_i is withdrawn from the unminimized target page and remaining entries are reminimized and updated in the TCAM. Also, all the page table identities corresponding to the target page are recomputed and updated in the page table.

4.3 Incremental Update Performance

The execution time of and insertion/deletion algorithm includes the time taken to search the covering page identity, reminimize the target page, and update the corresponding TCAM pages and page table. However, most of the time is spent on reminimization due to the exponential behavior of the Espresso-II, the algorithm used for minimization. The execution time to minimize the prefixes using the proposed approach and the one due to the technique used in [9] is shown in Table 3. The size parameter in Table 3 represents the maximum input that would be given to the Espresso-II algorithm for minimization during a prefix update. The Espresso-II algorithm was run on an Athlon dual processor running at 1.6 GHz to minimize the prefixes[1] when a request for updating a new prefix is issued by the neighboring router using the approach in [9]. The worst-case incremental prefix update took 63.04 sec for the `bbnplanet` router and 1,098.47 sec for the `attcanada` router, which appear to be expensive for high-performance routers. The proposed approach, on the other hand, only takes 0.006sec for incremental prefix update. The value is not only small and practical, but also bounded since, at any point of time, the number of inputs to the Espresso-II algorithm never exceeds 256. The proposed approach only uses 8 bits prefix length for minimization as opposed to 32 bits in [9] and, hence, makes the computation fast.

1. Prefixes here indicate the routing table prefixes after initial compaction (prefix overlapping and minimization).

```
Deletion(Pi)
If( |Pi| < w1)
DeleteFromBucket(Pi)
Else
If(∃ Ci that covers Pi)
      Find Ci such that
      |LCS(PCover(Ci))|=|LCS(Pi)|
      P=Cover(Ci)
      P=P-Pi
      Q=Minimize(P)
      DeleteFromTCAM(Q'⋂P)
      InsertToTCAM(Q⋂P')
      UpdatePageTable(Ci)
Else
      DeleteFromBucket(Pi)
```

Fig. 10. Algorithm 4: Incremental Deletion.

4.4 Memory Management

During the initial configuration, the comparators, page tables, and pages are set up based on the optimal architectural parameters. The design of the router is based on the prefix statistics and, hence, it is less likely that the pages will overflow frequently. The page fill factor γ ensures that, during the build time of the router, the pages have enough space for future updates. Also, most of the prefix updates are route flaps. So, it is more likely that the same set of prefixes will be added and deleted that would result in much less chance of an overflow. However, it is still possible that the pages could overflow. In the previous section when we discussed the insertion algorithm, we did not consider the possibility of an overflow. Hence, we present a memory management technique that will effectively reorganize the pages upon pages overflow without affecting the update time. Let P_i be the prefix that resulted in the overflow.

The memory management algorithm ensures that only the pages that have their page identity the same as C_i are recomputed during an overflow. It is important to note that, if an overflow results in creating an extra page, the free pages already present in the TCAM will be used and the page tables will be updated appropriately. The memory management technique triggers a reconfiguration, according to Algorithm 5 (shown in Fig. 11), if the page table overflows.

The insertion and deletion of prefixes could also result in page fragmentation. To overcome this problem, all the pages that have the same page ID have prefixes sorted in increasing order of prefix length. We then adopt the

TABLE 3
Comparison of Incremental Update Performance

Router	Total Prefixes	Approach in [9]		Proposed Approach	
		Size	Time (sec)	Size	Time (sec)
attcanada	112412	15146	1098.47	223	0.005
bbnplanet	124538	7580	63.04	256	0.006

technique discussed in [7] to efficiently insert and remove entries within the TCAM to reduce such fragmentation.

5 Results and Case Studies

In this section, we present the results based on the proposed architecture. The results have been illustrated with two real-life large routers (bbnplanet and attcanada) to demonstrate the effectiveness of the proposed approach. We also compute the range of w_1 for which power is bounded and memory requirement is the least. Finally, we evaluate the results corresponding to above two router examples.

5.1 Power Consumed per Lookup

The power consumption in TCAM architecture is determined in terms of the number of active TCAM entries that are enabled during the lookup operation. The number of active entries in the architecture depends on the selection of parameter w_1. We also validate the choice of $w_1 = 8$ made in Section 3.

We know that power consumption is also proportional to the number of entries looked up in comparators, page table, and the bucket. As discussed earlier, the number of entries in the comparator and page table is constant and negligible for a particular value of w_1. However, the number of active pages and buckets during the lookup operation depends on the match/mismatch in the first level comparator TCAM. Thus, the bound on maximum power consumption per lookup depends on the size of the bucket and the number of active pages looked up in the second level TCAM. From Lemma 3, we know that the number of entries looked up in the pages is a constant. In order to find all the values of w_1 for which power is bounded, we examine all values of w_1 for which the bucket size is less than page size. For those values of w_1, we can confirm that the power is bounded by the number of pages enabled in the second level $(256 * 3)$. We have used α_f as the fill-factor to bucket design and considered its value as 0.5 in our simulation to ensure that 50 percent of the bucket space is available for future updates. We have also used a fill factor (γ) of 0.5 for the pages in our page filling heuristics.

```
Let Qw1 = pi1pi2pi3..piw1
For all Pi such that Qw1 covers Pi
     FillPages(P,w1, βmax,1,C)
     UpdatePageTable(Ci)
     ReprogrammeDeMux(Ci)
```

Fig. 11. Algorithm 5: Memory Management Heuristics.

We plot the number of active entries in the bucket during lookup operation for varying w_1 and keeping the page entries to its bound. We have used the modified version of CACTI-3.0 [15] to analyze the approximate power consumption in TCAM. The CACTI-3.0 model was developed for analysis of cache memories, so we modified the source code of a fully associative cache to model the behavior of CAM. This approach is also used in a prior study [14] as CAM operation is equivalent to tag matching in a fully associative cache. We use 0.18um process technology parameters and considered the operating frequency of 100 MHz to evaluate the power consumption in CAM. Fig. 12 and Fig. 13 show the power consumption results for the bbnplanet and attcanada routers, respectively. It is evident from these figures that, for $w_1 \leq 13$, we obtain a bound on the power consumption for both the routers. This also validates the architectural choice of $w_1 = 8$. It may be worthwhile to mention that we have not shown the values for $w_1 > 16$ since, for such values of w_1, the power consumption per lookup can be very high and will not be useful to router design.

Now, we compare the worst-case bound on power consumption between the proposed architecture and the techniques given in [10]. The two techniques used in [10] are bitmapping and partitioning. The worst-case bounds for bitmap and partition techniques are $O(C_{max} + P_b)$ and $O(N/b + .S_d)$, respectively. In the bitmap technique, though the number of entries enabled during the search is very small (C_{max}), the entries corresponding to prefixes of length less than 16 and greater than 24 are always enabled during every lookup operation (P_b). The value of P_b is very large (65,068) for the bbnplanet router. In the partition technique, the power consuming components consist of index TCAM and data TCAM. Only a page of size N/b is enabled in the data TCAM, where b is the number of pages in TCAM. However, the entire index TCAM of size S_d is searched for every lookup, which is large if the page size is small. If the page size is large, the index TCAM has a small number of entries; however, the page enabled in data TCAM contributes mainly to the power consumption. In our approach, the worst-case bound is $O(S_i)$, where S_i is the size of the portion of the TCAM segment that is enabled and it is quite small compared to the entire TCAM array (maximum $256 * 3$). Since most of the segments are of very small sizes, the average power consumption is also quite small. It may be important to mention that the word length of TCAM in our architecture is 24 bits as opposed to 32 bits in conventional TCAMs.

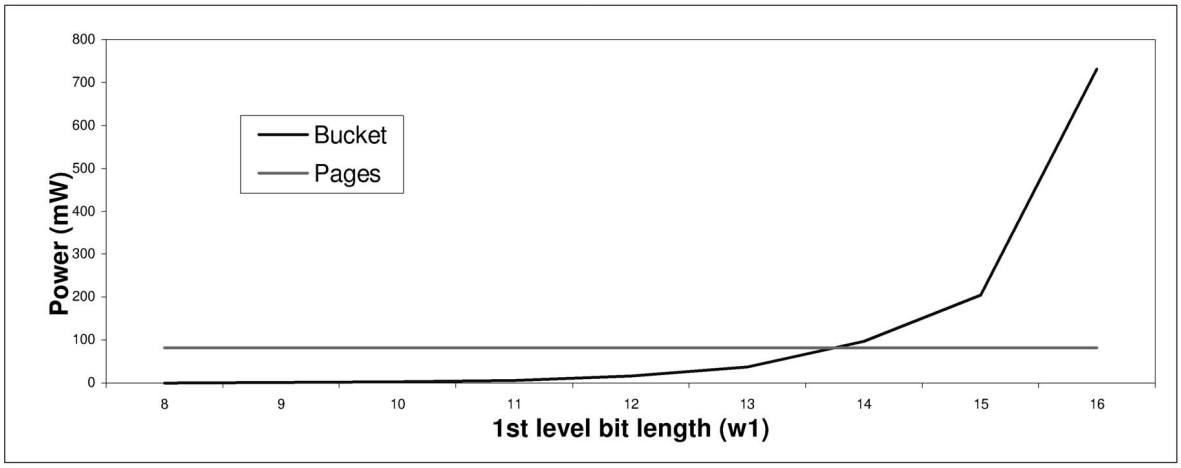


Fig. 12. Power consumption in the `bbnplanet` router.

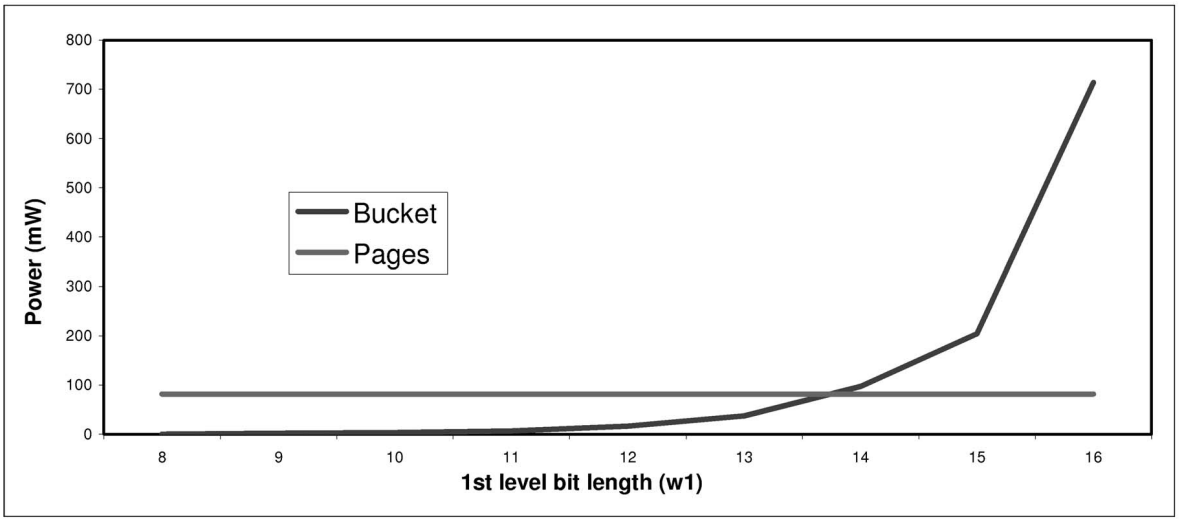


Fig. 13. Power consumption in the `attcanada` router.

5.2 Memory Requirements

The total memory requirements at different values of w_1 are given by (1). We use this equation to find the optimal value of w_1 for which power is bounded and memory requirement is the least. From the above discussions, we observe that, in the cases of `bbnplanet` and `attcanada`, power is bounded for $8 \leq w_1 \leq 13$ (Fig. 12 and Fig. 13). From the plot given in Fig. 14, we see that the values of w_1 for which the memory requirement and power consumed remains bounded are in the same range for `bbnplanet` and `attcanada` routers, respectively.

It is important to mention that we have not considered the value of $w_1 < 8$ in this paper as we observed that there is an increase in memory consumption when w_1 is decreased from 8 through 1, though the power requirement is bounded by the $256 * 3$ page entries.

5.3 Case Studies

Table 4 and Table 5 show the reduced memory requirement and power consumed in term of entries enabled for searching in both the routers. For these results, we applied compaction and architectural techniques successively without *fill-factors* ($\gamma = 1$ and $\alpha_f = 1$).

We see that about 48 percent compaction is obtained in the case of `bbnplanet` router and 44 percent compaction in the case of `attcanada`. Table 5 shows the power reduction as an equivalent to the number of TCAM entries. Column 3 in Table 5 indicates the reduction in the number of entries looked up due to the compaction and column 4 is due to the benefits from the proposed architectural techniques. The large routers like `attcanada` and `bbnplanet` can be designed with low power per lookup using the EaseCAM approach.

6 Related Work

The TCAM-based routing lookup has been proposed by several authors [2], [3], [5], [9], [16]. In [3], the authors discussed the use of TCAM in hardware search engine and called that Database Accelerator. The fast routing table lookup using TCAM was proposed in [5] wherein the authors used sorting of prefixes to table lookup. Subsequently, Shah and Gupta [7] provided techniques that avoid the overhead of sorting. Kobayashi et al. associated priority to each TCAM entry in [2] and showed the use of extra hardware for output for multiple prefix matches. Their

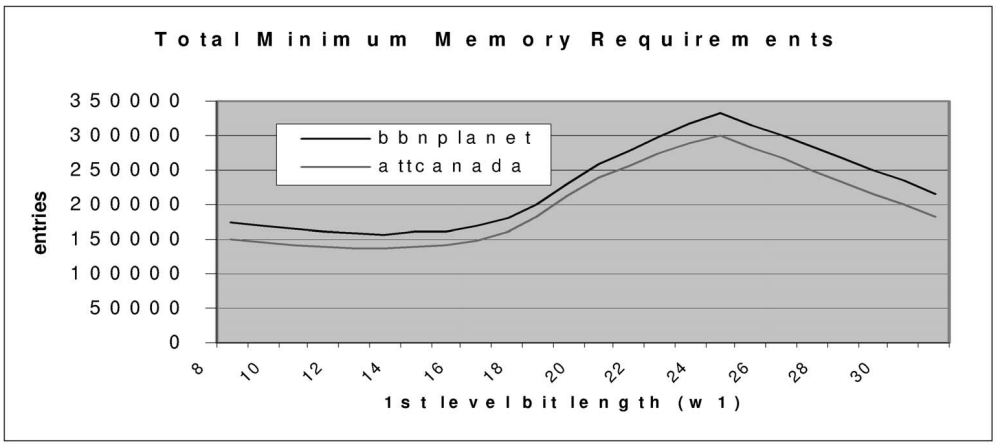


Fig. 14. Total memory requirements.

TABLE 4
Reduction in Memory Requirements

Router	Raw Data (No Of Entries)	After Compaction (No Of Entries)	Effect of Architecture (No Of Entries)
attcanada	112412	57837	50182
bbnplanet	124538	71500	59883

TABLE 5
Power Estimation Related to Number of Entries in TCAM-Based Router

Router	Raw Data (No Of Entries)	After Compaction (No Of Entries)	Effect of Architecture (No Of Entries)
attcanada	112412	15146	669
bbnplanet	124538	7580	768

approach avoids sorting requirements at the cost of increased latency due to the extra hardware. The authors in [1] have tackled the IP lookup problem using specialized hardware. The authors in [8] and [11] have proposed the compaction technique based on prefix pruning and showed significant prefix compaction. In [12], the authors have suggested an on-chip minimization algorithm for compacting prefixes. In [9], the authors proposed placing TCAMs on a separate bus for parallel access and introduced paged-TCAM architecture to increase the throughput and reduce the power consumption in TCAM routers. Recently, the authors in [10] proposed bit-selection architecture and partitioning technique to design power efficient TCAM architecture.

7 Conclusion

In this paper, we presented a novel architecture for a TCAM-based IP forwarding engine. We have shown significant reduction in memory usage based on the prefix compaction and architectural design. We designed a heuristic to store entries in TCAM pages so that only a bounded number of entries are looked up during the search operation. A fast incremental update scheme has been introduced that is time bounded. We also proposed an efficient memory management technique to tackle the overflow problem. The memory requirements and power consumption for router architecture have been outlined. To demonstrate the merit of the proposed architecture, we used the architectural features on six routers based on their trace statistics and evaluated the benefits of our approach. It has been shown that the memory requirement is reasonably low due to the use of effective compaction technique. At the same time, the power consumption is found to be remarkably low to promise efficient TCAM design in the future.

Acknowledgments

The authors sincerely thank Seraj Ahmad at Texas A&M University for his help in providing additional simulations and reviewing the manuscript during the revision process.

References

[1] P. Gupta, S. Lin, and N. McKeown, "Routing Lookups in Hardware at Memory Access Speeds," *Proc. INFOCOM*, pp. 1240-1247, Mar. 1998.
[2] M. Kobayashi, T. Murase, and A. Kuriyama, "A Longest Prefix Match Search Engine for Multi-Gigabit IP Processing," *Proc. Int'l Conf. Comm.*, June 2000.
[3] J.P. Wade and C.G. Sodini, "Dynamic Cross-Coupled Bit-line Content Addressable Memory Cell for High Density Arrays," *IEEE J. Solid-State Circuits*, vol. 22, no. 2, pp. 119-121, Feb. 1987.
[4] R.K. Brayton et al., *Logic Minimization Algorithms for VLSI Synthesis*. Boston: Kluwer Academic, 1984.
[5] A.J. McAuley and P. Francis, "Fast Routing Table Lookup Using CAMs," *Proc. INFOCOM*, pp. 1382-1391, Mar. 1993.
[6] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Computer Systems*, vol. 17, no. 1, pp. 1-40, Oct. 1999.

[7] D. Shah and P. Gupta, "Fast Updates on Ternary-CAMs for Packet Lookups and Classification," *IEEE Micro,* vol. 21, no. 1, Jan./Feb. 2001.
[8] R. Panigrahy and S. Sharma, "Reducing TCAM Power Consumption and Increasing Throughput," *Proc. IEEE Hot Interconnects,* Aug. 2002.
[9] H. Liu, "Routing Table Compaction in Ternary-CAM," *IEEE Micro,* vol. 22, no. 1, Jan./Feb. 2002.
[10] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: Power-Efficient TCAMs for Forwarding Engines," *Proc. INFOCOM,* Apr. 2003.
[11] M.J. Akhbarizadeh and M. Nourani, "An IP Packet Forwarding Technique Based on Partitioned Lookup Table," *Proc. Int'l Conf. Comm.,* Apr. 2002.
[12] R. Lysecky and F. Vahid, "On-Chip Logic Minimization," *Proc. ACM/IEEE Design and Automation Conf. (DAC),* June 2003.
[13] G. Huston, "Analyzing the Internet's BGP Routing Table," *The Internet Protocol J.,* vol. 4, 2001.
[14] M. Zhang and K. Asanovic, "Highly Associative Caches for Low-Power Processors," *Proc. IEEE MICRO-33,* Dec. 2000.
[15] P. Shivakumar and N.P. Jouppi, "Cacti 3.0: An Integrated Cache Timing, Power and Area Model," Western Research Lab (WRL) Research Report, 2001/2002.
[16] Z. Dai and B. Liu, "A TCAM Based Routing Lookup System," *Proc. 15th Int'l Conf. Comm.,* pp. 1090-1096, 2002.

V.C. Ravikumar received the MS degree in computer science from Texas A&M University in May 2004. He is currently working as a system designer at Hewlett Packard. His research interests are networking, operating systems, and embedded systems.

Rabi N. Mahapatra received the PhD degree in computer engineering from the Indian Institute of Technology at Kharagpur, India, in 1992. He was an assistant professor in the Electric and Electrical and Computer Engineering Department at IIT Kharagpur until 1995. Currently, he is an associate professor with the Department of Computer Science at Texas A&M University, College Station. His current research interests include embedded system codesign, system-on-chip, VLSI design, and computer architectures. He has published more than 55 papers in reputable journals and conference proceedings. His recent publications can be found at http://faculty.cs.tamu.edu./rabi/. He is a senior member of the IEEE.

Laxmi Narayan Bhuyan has been a professor of computer science and engineering at the University of California, Riverside since January 2001. Prior to that he was a professor of computer science at Texas A&M University (1989-2000) and program director of the Computer System Architecture Program at the US National Science Foundation (1998-2000). He has also worked as a consultant to Intel and HP Labs. His current research interests are in the areas of network processor architecture, Internet routers, and parallel and distributed processing. He has published more than 100 papers in related areas in reputable journals and conference proceedings. His brief biography and recent publications can be found on his Web page at http://www.cs.ucr.edu/~bhuyan/. He is a fellow of the IEEE, a fellow of the ACM, and a fellow of the AAAS.