A Memory- and Time-efficient On-chip TCAM Minimizer for IP Lookup

Heeyeol Yu

Computer Science & Engineering Department
University of California, Riverside
Email: hyyu@cs.ucr.edu

Abstract—Ternary content addressable memories (TCAMs) are becoming very popular due to the simple-to-design IP lookup units included in high-speed routers; they are fast and simple to manage, and they provide a one-clock lookup solution. However, a major drawback of TCAM-based IP lookup schemes lies in their high power consumption. Thus, the rapid increase of routing tables inevitably deteriorates TCAM power efficiency. Although on-chip TCAM minimizers aim for the TCAM power efficiency in a fast time and at a small memory amount, the minimizers are not efficient in a large scale prefix table. In this paper, we present a hash-based on-chip TCAM minimization for a power- and throughput-efficient IP lookup. In a hash-based TCAM minimization (HTM), we convert prefixes into keys and merge keys with a fast hash lookup in an $O(nW)$ complexity, where $n$ is the number of prefixes and $W$ is the number of IP bits. Additionally, by building a forest of merging trees and choosing a subset among them, we can achieve a higher minimization ratio. The simulation with two routing tables shows that our HPM scheme uses 8.6 and 4.0 times fewer computation time and memory, compared to a contemporary on-chip minimizer.

I. INTRODUCTION

By definition, the ternary content addressable memories (TCAMs) are fully associative memories that allow a “don’t care” state to be stored in each memory cell in addition to Os and 1s. In particular, this feature attracts route lookup applications since they require the longest prefix matches. When a destination IP address is presented to a TCAM, each prefix TCAM entry is looked up in parallel, and the longest prefix that matches the address is returned. Thus, a single one-clock TCAM access is sufficient to complete an IP lookup operation.

A. Motivation

The demand for high-speed and large-scale routers continues to surge in networking fields. Since a fast packet forwarding is a router’s critical data path, it involves three major techniques: trie- [1, 2], hash-based [3, 4], and TCAM schemes [5, 6]. These techniques do have known limitations, such as $O(W)$ lookup complexity, where $W$ is the number of IP bits, and pointer overhead in trie-based schemes and a singleton match in hash-based schemes.

Unlike the SRAM-based (i.e. trie and hash) schemes with the inherent limitations, the TCAMs have become the de facto industrial standard for a high speed IP lookup and it is reported that there were more than 6 million TCAM devices deployed worldwide in 2004 [7]. However, despite their one-clock high-speed IP lookup, the TCAMs have the following limitations to overcome: low cell density, high power consumption, and high hardware cost. 1) Low cell density: in general, an SRAM cell consists of 6 transistors. However, a TCAM cell requires two SRAM cells for a “don’t care” state and a comparison logic which are made of 16 transistors in total. 2) High power consumption: since a TCAM entry match drives a high voltage in advance and it conducts a parallel search in TCAM entries for an IP lookup, consequently the TCAM chips demand higher power consumption. Currently, the high-density TCAM chips consume as much as 12 to 15 Watts each when all the entries are enabled for the parallel search. 3) High hardware cost; the TCAMs are expensive not only due to their low memory cell density but also due to their insignificant market demand which means they are not mass produced to drive their cost down. Also, the high power consumption incurs additional expenditures when factoring in other equipments, such as power supplies and cooling costs.

B. The Problem

The aforementioned TCAM problems have been the primary issues that the router industry and academia have strived to overcome. To name a few, an author in [8] propose a seminal scheme to minimize an IP prefix table, and a literature on TCAM logic minimization propose efficient methods to provide a fast TCAM minimization process and a reduced routing table size [6, 8–10]. Since the logic minimization is known to be an NP-complete problem, thus exact algorithms, like Espresso-II [11], require the off-chip workstation environment due to their large computing and memory requirements. Fig. 1 shows two minimization approaches: off-chip and on-chip minimizers. Whenever a network process needs a minimized routing-table TCAM, it asks the minimization load to a workstation with a high computation power.

However, an off-chip logic minimizer scheme requires a communication in between and this communication overhead can greatly slow the optimization process. This slow logic minimization performance is not suitable for current backbone routers where they require more than 290K prefixes. Alternatively, a logic minimizer can be performed on-chip where the minimizer and a router’s applications share the same processing resources or the minimizer uses a separate embedded processor/memory system.
Authors in [9, 10] propose on-chip TCAM minimizers to overcome the off-chip schemes’ overhead by using a small data and a fast execution time within a given limited resource budget. However, these on-chip schemes still suffer from a large data size and a slow execution time with the loss of minimization accuracy due to a localized minimization heuristics. For instance, a node in the recently-proposed minimization trie [9] needs three pointers and the number of nodes is bounded by $O(nW)$. In addition to memory overhead, a traversal from a root to a leaf requires a number of memory accesses that are bounded by the minimization trie’s height, i.e., $O(W)$. Lowering memory and time bounds, like $O(nW)$ and $O(W)$ in the minimization trie, is necessary for an on-chip minimizer in a scalable IP lookup demand.

C. Our Solution

We have seen that the heuristic TCAM minimizers in [9, 10] suffer from time and memory overheads which are not suitable for an on-chip minimizer. As a solution, we introduce a hash-based $O(nW)$ TCAM minimization (HTM) as an on-chip minimizer for a scalable IP lookup, where $n$ is the number of prefixes and $W$ is the number of IP bits. In an HTM, we convert prefixes into keys and merge two keys by using a hash lookup whose time complexity is $O(1)$.

This paper has the following contributions:

- We propose hash-based $O(nW)$ TCAM minimization for a power-efficient IP lookup, and these time- and memory-efficient schemes are suitable for an on-chip minimizer.
- In an HTM, we propose $O(W)$ one-symbol prefix merging and $O(nW)$ sub-optimal TCAM minimizer through prefix-to-key transformation.
- In IP lookup simulation, an HTM achieves on average 8.6 and 4.0 times computation time and memory saving.

The rest of the paper is organized as follows. Sec. II provides the IP lookup background and existing TCAM minimization schemes. In Sec. III, we introduce a hash-based $O(nW)$ TCAM minimization. Finally, after we present simulation results in Sec. IV, we make a conclusion in the following Sec. V.

II. BACKGROUND AND RELATED WORK

A. IP Lookup Schemes

The three representative IP lookup schemes of trie, hash, and TCAM have different lookup complexities of $O(W)$, $O(1)$, and $1$, respectively, in terms of clock cycle. Although each scheme has its own pros and cons, a TCAM scheme provides a one-clock IP lookup solution and this is preferred by the router industries due to its easy-to-implement router design and the worst case lookup support.

For a given routing table, we load the prefixes into a TCAM in a decreasing prefix-length order. Through this descending order, an entry with the smallest index ensures the longest matched prefix for a destination IP address as it is searched through several matched TCAM entries. By using this index, we can access the SRAM word where the next-hop associated with the matched prefix is stored, and we complete the forwarding task. Fig. 2 shows the TCAM-based IP lookup with three prefixes. In the figure, a packet destination IP is compared with every prefix entry in parallel and in one clock. Although there are two matched entries, a priority encoder hardware chooses the entry with the smallest index and we access an SRAM word with the same index for the next-hop.

Although two accesses to TCAM and SRAM for an IP lookup reasonably simplify an IP lookup design, a tremendous amount of power consumption and the TCAM cost deter router vendors from using TCAM-based schemes yet as the issues of high-speed lookup and scalability emerge in the near future. Thus, the following TCAM logic minimization solution is necessary for providing a high power efficiency.

B. Minimizations of Boolean Logic and TCAM

Boolean logic minimization schemes traditionally find their use in logic synthesis in order to reduce the number of logic gates for a given circuit. The exact solution to a logic minimization problem is given by Quine [12] and McClusky [13]. Studies have suggested that Quine-McClusky procedure and its variants are inefficient because they generate a huge number of primes, i.e. the sum-of-products (SOPs) which are defined to be independent SOPs not contained in another SOP.

In IP lookup domain, Liu in [8] first proposes the routing table compaction using a logic minimization technique. After this seminal approach is published, authors in [6, 9, 10] propose an efficient logic minimizers designed for conducting the IP lookup. In a logic minimization application to IP lookup, router vendors favor an on-chip minimizer which
shares the same processing resource with router applications since the on-chip minimizer provides a very fast solution at the cost of a sub-optimal TCAM minimization [9, 10]. However, our scheme does not waste a significant memory and time for a TCAM minimization like these on-chip minimizers. For instance, a minimization trie, M-trie, in [9] uses a tree data structure that requires a pointer overhead since each node in the tree consumes three pointers to its children. In addition to the memory overhead, our scheme takes \( O(W) \) time complexity due to a linked-list-based hash data structure and a minimization trie for one prefix takes \( O(W^2) \) since traversing the tree from a root to a leaf takes \( O(W) \). While the number of hosts is tripling every two years [14], the routing table size has rapidly increased from 50K to 290K in 10 years [15]. Thus, in this unprecedented increase, the existing minimizers hardly meet the time scalability while our on-chip minimizer does.

### III. A Hash-based TCAM Minimization

This section introduces two schemes: a one-symbol merging for a fast TCAM minimization and a merging time complexity analysis.

In a boolean logic minimization, a boolean function \( F \) with \( \ell \) input variables, \( a_1, \ldots, a_\ell \), is defined as \( F : B^\ell \to B \) where \( B=\{0,1\} \). Thus, given an input \( a=[a_1, \ldots, a_\ell] \in B^\ell \), \( F \) returns 0 or 1. The on-set \( S_{on} \subseteq B^\ell \) is the set of input \( a \) values such that \( F=1 \) while the off-set \( S_{off} \subseteq B^\ell \) is the set of input \( a \) values such that \( F=0 \). A two-level logic minimization involves finding a minimum covering set \( G \) for the specified function \( F \) such that the set \( G \) contains the SOP representation of all the input variables which set the function \( F \) to a value of 1. Since finding such a \( G \) is \( NP \)-complete [16], we present a heuristic scheme in a tradeoff between the optimal covering set and memory- and time-efficiencies in the following sections.

#### A. One-symbol Prefix Merging

In a TCAM minimization, we want to minimize the prefix set of the same prefix length and the same next-hop so that we can reduce TCAM power consumption. In such a TCAM minimization, \( a_i \) corresponds to a \( i \)-th prefix bit and \( \ell \) becomes a prefix length as does in a boolean logic minimization. Authors in [9] represent a prefix as a path in a minimization trie and merge two prefixes of the same length and the same next-hop into a new prefix.

Unlike the M-trie scheme [9] with overheads of pointer memory and bit-by-bit traversal time, we strategically use a hash-based merging in a TCAM minimization. For our hash operation, we introduce a prefix-to-key transformation which converts a prefix notation into a key notation. Suppose a prefix \( P \) is a sequence of \( \ell \) symbols, which means it is a member of \( \{0,1,X\}^\ell \). We use two bits to represent a symbol in \( P \) so that the prefix \( P \) of \( \ell \) symbols becomes a key of \( 2\ell \) bits. Table I shows the detailed prefix symbol representation. For instance, a prefix 1X1 becomes a key of 1000102 bits whose decimal value is 34.

In this prefix-to-key representation, we conduct a basic one-symbol merging between two prefixes as follows: Given a prefix \( x_1X_2x_3 \in \{0,1\} \) and \( s_1, s_2 \in \{0,1,X\}^\ell \), we can convert it to a key \( k \), find a corresponding key \( k' \) of prefix \( s_1X_2s_3 \), through a hash lookup, and merge two keys \( k \) and \( k' \) into a key \( k'' \) of a prefix \( s_1X_2 \). For the hash lookup, we convert all prefixes into keys with help of Table I, and we conduct a hash lookup to find the corresponding key among an \( n \)-key pool in \( O(1) \) time complexity. Note that for one-symbol merging we need to have one hash lookup and one hash insertion if found: 1) one hash lookup of key \( k \) takes \( O(1) \) on average [17], 2) one hash insertion of key \( k' \) in \( O(1) \) time complexity if we find the key \( k' \). Since the prefix length of concern is \( W \), we perform \( W \) times one-symbol merging from the first symbol to the last \( W \)-th symbol. Also, once we get a new merged key \( k'' \), we delete these old \( k \) and \( k' \) and freshly conduct another one-symbol merging on the new key \( k'' \).

![Fig. 3. One-symbol merging for prefixes \( P_1, \ldots, P_6 \) and corresponding keys](image-url)

Fig. 3 shows such a one-symbol merging among 6 prefixes, \( P_1, \ldots, P_6 \), in a brute-force way. Suppose that we scan the six prefixes from \( P_1 \) and that we start to look at the first symbol for a one-symbol merging in each prefix. Since \( P_1 \) and \( P_2 \) have the last 3 common symbols 001, we can merge them into a new prefix \( P'_1 \) in a solid-lined round box as depicted in Fig. 3. Likewise, prefixes \( P_3 \) and \( P_4 \) merge together into \( P'_2 \) since they have common symbols 110. Since we delete the old prefixes \( P_1, \ldots, P_4 \) during a one-symbol merging, we only proceed to merge ones with active prefixes in solid-lined round boxes, not inactive prefixes in dash-lined round boxes. Algorithm 1 one-symbol–merging explains the detailed one-symbol merging.

In the algorithm, we can process Lines 6 through 11 in a fast CPU speed due to the bit operation while Line 12 needs memory accesses whose time complexity is \( O(1) \) on average due to the hash lookup. In contrast, an M-trie scheme in [9] needs \( O(W) \) memory accesses on average since an M-trie’s traversal is correspondent to memory accesses and is directed.

<table>
<thead>
<tr>
<th>Symbols in a prefix</th>
<th>( 0 )</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key bits</td>
<td>01</td>
<td>10</td>
<td>00</td>
</tr>
</tbody>
</table>

**TABLE I**

**Prefix symbol representation in two bits**
Algorithm 1: one-symbol-merging

Input: a prefix set $S$ of the same length $\ell$ and next-hop
Output: a set of merged prefixes
1 Convert all prefixes to keys according to Table I;
2 Save the keys in a hash table $H$: // $|H|=n$, key size
for each key $k \in H$ do
  for $i=1$ to $\ell$ do
    mask1=112; // 2-bit mask
    value = $k \gg (2^{\ell}(i-))$;
    if value == 102 then value = 012;
    else if value == 012 then value = 102;
    else if value == 002 then continue;
    mask2= (mask1 << $(2^{\ell}(i-))$);
    $k' = (k \& \text{ 'mask2')} | \text{ value}$;
    hash lookup of $k'$ in $H$: // $O(1)$ time
    if found $k'$ in $H$ then
      hash deletion of $k$ and $k'$ in $H$: // $O(1)$ time
      value = 002 << $(2^{\ell}(i-))$;
      $k'' = (k \& \text{ 'mask2')}$;
      hash addition of $k''$ in $H$: // $O(1)$ time
  end
end

by $W$ prefix bits. In conclusion, our algorithm takes $O(nW)$
time complexity due to the loop in Lines 3 and 4 while an
M-trie scheme takes $O(nW^2)$. Since the gap between the CPU
processing time and the memory-access time is in an order
of magnitude [18], our hash-based merging has a significant
benefit as shown in the following memory-access analysis.

B. Merging Time Complexity Analysis
Suppose that a linked-list-based hash table $T$ has $m$ buckets
and its hash function $h(\cdot)$ whose range is $[1, \cdots , m]$ and that $N_u$
is the number of memory accesses for an unsuccessful search
in the hash table. Under the assumption of simple uniform hashing, any key $k$ not already stored in the table is equally
likely to be hashed to any of the $m$ buckets [17]. The expected
number of memory accesses to unsuccessfully search for a key
$k$ among $n$ keys is the expected linked list length in searching
to the end of list $T[h(k)]$ as follows:

$$E[N_u] = E[L_{h(k)}] = \alpha$$   \hspace{1cm} (1)

where $L_{h(k)}$ is the length of a linked list indexed by $h(k)$ and
load factor $\alpha=n/m$. Note that we needs an additional memory access, which we will consider in Eq. (3), in order to detect
the starting pointer of a linked list in a hash table bucket.

In addition to an unsuccessful search, we derive the number of
memory accesses, $N_s$, in a successful search as follows: We
assume that a key being searched for is equally likely to be
any of the $n$ keys stored in the hash table. The number of
keys examined during a successful search for a key $k$ is 1
more than the number of keys that appear before $x$ in $x$'s
linked list, under the condition that new keys are placed at the
front of their linked list. Thus, to find the expected number of

days paradox [17]. Thus, the expected number of keys exam-
ined over the $n$ keys in the hash table, we take the
average of 1 plus the expected number of keys added to $k$’s
linked list after $k$ is added to this linked list. Let $k_i$ denote
the $i$-th key inserted into the table, for $i = 1, \cdots , n$. For keys $k_i$
and $k_j$, we define the indicator random variable $X_{ij}$=$I(h(k_i)=h(k_j))$.
Under the assumption of simple uniform hashing, we have
$Pr(X_{ij})=Pr(h(k_i)=h(k_j))=1/m$, and $E[X_{ij}] = 1/m$ by a birthday
paradox [17]. Thus, the expected number of keys examined in a successful search is the following:

$$E[N_s] = E[1 + \sum_{j=1}^{n} X_{ij}] = 1 + \frac{1}{nm} \sum_{j=1}^{n} \sum_{i=1}^{n} E[X_{ij}]$$

$$= 1 + \frac{1}{nm} \left( \sum_{i=1}^{n} \left( \frac{n(n+1)}{2} \right) \right) = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$$   \hspace{1cm} (2)

where the load factor $\alpha=n/m$.

Eqs. (1) and (2) explain the average number of memory accesses
for unsuccessful and successful searches, respectively, regardless of a symbol position $i$, $1 \leq i \leq W$. Thus, under the condition that a one-symbol merging occurs at symbol position $i$, then position $r, 0 < r < i$, involves unsuccessful searches taking $E[N_u]$ memory accesses and the $i$-positioned merging involves $E[N_s]$ memory accesses. If we unconditionalize this condition, we can derive the following average number of memory accesses in Algorithm 1:

$$E[N] = \sum_{i=1}^{W} E[N_s | B = i] \cdot Pr[B = i]$$

$$= \sum_{i=1}^{W} \left( \sum_{j=i}^{\ell} \left( 1 + E[N_u] \right) + 1 + E[N_s] \right) \cdot Pr[B = i]$$   \hspace{1cm} (3)

where a random variable $B$ is a bit position that triggers a one-
symbol merging. In this equation, we consider an additional
memory access for a hash bucket that contains a starting linked list for both unsuccessful and successful searches as in $1+E[N_u]$ and $1+E[N_s]$.

Fig. 4. The average number of memory accesses for an M-trie and our HTM
B for both the M-trie and our HTM in two hypothetical ways: the equal and skewed probabilities of $Pr(B=i)$. In the equal probability, we assume that each symbol position has an equal chance of merging with others. That is, $Pr(B=i)=1/32$. In contrast, in the skewed probability we set $Pr(B=i)<Pr(B=j)$, $1 \leq i < j \leq 32$. In both cases, the gap between M-trie and our scheme is noticeable in Fig. 4. Also, if we sum Y axis values, the difference of both schemes’ $E[N]$ becomes significant. An author in [19] reports that the prefix distribution is heavily centered from the population of prefix length 19 through 24. Thus, the skewed probability case is relatively more practical, and in this probability case our HTM’s $E[N]$ is 30.2 which is 7.5 times smaller than an M-trie’s $E[N]$. In addition to the time complexity benefit, our HTM scheme also achieves a memory saving benefit that a merged prefix only needs a data structure for a corresponding key and a pointer in a linked-list-based hash table. Note that the M-trie scheme needs $O(W)$ nodes of 3 pointers for the same merged prefix due to a path from a root to a leave. We will present the detailed memory and time complexity in Sec. IV.

IV. THE EXPERIMENTAL RESULTS

In this section, we present the performance comparisons of our schemes against two recently-published schemes: ROCM in [10] and M-trie in [9]. We used AS65000 and AS6447 of 290490 and 297052 prefixes from [15] in this experiment. Our hardware configuration was a 2.66GHz Intel Core i7 with a 1.06GHz DDR3 SDRAM.

A. The Minimization Comparison

Fig. 5(a) shows the number of merged prefixes as we run the three minimization schemes. ‘M-trie’ shows on average 5% better minimization performance than ‘ROCM’ while our scheme marked as ‘HTM’ shows the same performance as ‘ROCM’. Fig. 5(b) shows a power consumption which we measured with a TCAM modeling tool [20]. Note that we normalized each scheme’s power consumption to a power consumption without any minimization scheme. This figure shows a benefit of power saving in a TCAM minimization and it also indicates that our scheme is as good as ‘M-trie’ in terms of minimization performance and power saving.

B. The Time and Memory Comparison in Minimization

Note that each minimizer scheme uses its own data structure and this data structure determines its own memory-access time complexity in the minimization operation. For example, an M-trie uses a tree structure where a node has three pointers, and since a traversal from a root to a leaf takes $W$ memory-access time, a significant time difference is expected as in Fig. 4. Fig. 6 shows such time and memory differences in a consumed time and data structure memory used by minimizer schemes. Our HTM scheme achieves on average 1.9 and 4.0 times in memory saving over ROCM and M-trie, respectively. Furthermore, the HTM scheme achieves on average 403.7 and 8.6 times computation time saving. This noticeable time difference value of 403.7, that is between ROCM and our HTM, clearly indicates that the ROCM on-chip minimizer is not suitable for a scalable routing table.

C. The Complexity Comparison

Table II shows the complexities of memory access time and memory size between contemporary schemes and ours. Note that we consider the main memory access as one time unit in the time complexity analysis. Also, we assume that a 32-bit pointer is one memory unit in the memory complexity analysis. From this table, it is clear that time and memory complexities of our minimization scheme are smaller than those in an M-trie scheme.
V. CONCLUSION

Although a TCAM is widely deployed due to its design simplicity, its noticeably high power consumption does prohibit router vendors from utilizing it in the foreseeable future. On-chip TCAM minimizers, like ROCM and M-trie, aim to reduce TCAM sizes, and operate in an embedded system where CPU time and memory resources are limited. However, their scalability becomes issue as the routing table size increases rapidly.

In order to solve the TCAM scalability issue, we have proposed a hash-based TCAM minimization scheme to scalably look up an IP address, and the scheme is suitable for an on-chip operation due to its time- and memory-efficiencies. In the hash-based TCAM minimization, we transform prefixes into keys and utilize an $O(1)$ hash lookup to minimize prefixes in $O(nW)$ memory-access time complexity. Also, the memory complexity to operate our scheme is $O(n)$ due to a hash table.

We use two large-sized routing tables to test our scheme in the simulation. On average, our TCAM minimization scheme achieves on average 8.6 and 4.0 times in computation time and memory saving, compared to contemporary schemes. As a future plan, we will extend our TCAM schemes into the packet classification domain.

VI. ACKNOWLEDGEMENT

This work is partly sponsored by Samsung and Google Korea.

REFERENCES