A memory-efficient parallel routing lookup model with fast updates

Yanbiao Li, Dafang Zhang, Kun Huang, Dacheng He, Weiping Long

Article info
Article history:
Received 17 April 2013
Received in revised form 7 October 2013
Accepted 14 October 2013
Available online 30 October 2013

Keywords:
Split Trie
Memory efficient
Fast update
Parallel lookup

ABSTRACT
Routing lookup, as a core function of routers for forwarding and filtering packets, has confronted with serious challenges nowadays, ranging from memory efficiency, update performance and throughput. Rather than seeking optimization techniques for the traditional lookup model, this paper presents a brand-new parallel lookup model, named Split Routing Lookup Model. In consideration of partial similarities among prefixes, we split all prefixes to produce redundancies, which are then removed during information integration. After that, the on-chip structure is compressed sharply. Besides, by such “splitting”, route updates are diverged to be more targeted, and the lookup process is also decomposed to support parallel processing.

With 14 real-world routing data, the proposed model is evaluated through 4 classic trie-based approaches, in comparison with their traditional implementations. The encouraging results show the superiorities of the proposed model in a comprehensive view. The on-chip memory savings are up to 99.2% and 94.8% for IPv4/6 respectively. While the reduction of update overhead, even in the worst case, is 50% and 30% respectively. Moreover, the pipeline depth is also reduced by 25–50%. Besides, another 2 techniques are selected to evaluate the proposed model on the virtual router platform. According to the results, based on the proposed model, 160 KB on-chip memory is enough to store 14 virtual routers, each consuming only 11 KB on average. In this way, the scalability of the proposed model to virtual routers is also clearly demonstrated.

1. Introduction
Routing lookup, namely IP address lookup for IP routers, is one of the key functions of high-performance routers for packets forwarding and classifying. With the rapid development and large-scale use of emerging network technologies and applications, the link rate is expected to reach higher and higher level in near future, which poses a strong demand on improving the performance of routing lookup.

On the other hand, a backbone router, nowadays, has to hold a large routing table with more than 400K entries [1]. The gap between the ever-increasing size of routing tables and the limited high-speed memory is widening. What’s worse, the memory limitation will become more serious for IPv6 and future network architectures, such as Named Data Networking (NDN) [2] and Software-Defined Networking (SDN) [3]. Consequently, routing lookup engine is facing enormous challenges over memory efficiency and scalability.

Furthermore, frequent updates may disrupt the lookup process and thus influence the throughput. To keep stable throughput under frequent updates, update overhead must be controllable.

Especially in virtual routers [4] and open-flow switches [5], the aggregates of route updates in each router instance, and the highly frequent updates over flow tables, all call for efficient update mechanisms.

1.1. Prior arts
Traditionally, routing lookup is modeled as a Longest Prefix Matching (LPM) problem over a specified Forwarding Information Base (FIB) (although multiple FIBs coexist in a virtual router platform, each lookup targets only one of them.). Classic solutions to LPM can be divided into three major categories.

Ternary Content Addressable Memory (TCAM)-based solutions provide deterministic and high-speed lookup (one lookup per cycle) [6,7]. But they can only handle small or middle-size tables due to their high cost and power. Besides, keeping records in order makes update more complicated. It is another key limitation of these solutions [8]. Hash-based solutions can achieve fast lookup with simple updates through novel hashing mechanisms [9–11]. However, their application is impeded by prohibitive requirements of high-bandwidth memory, false positive rates and hash conflicts. By contrast, Trie-based solutions are the most widely used ones because of their flexibility and expandability [12].
The key issue of trie-based solutions is that one lookup always requires multiple memory accesses. Multi-bit trie [13,14] reduces the worst-case lookup overhead with memory controlled, but it only resolve this issue to some extent. By mapping a trie onto a Static Random Access Memory (SRAM)-based pipeline, the average lookup speed of processing a sequence of requests is pushed to one lookup per cycle approximately [15–18]. However, seeking for a balanced pipeline architecture always requires a large number of stages [16,17,19]. The resulted low memory efficiency and long lookup latency are unadaptable to large tables and longer prefixes (such as for IPv6). Although this issue can be addressed through additional compressing techniques [15,18,19], update overhead becomes the next problem.

In the virtual router platform, multiple FIBs coexist in the same physical device. It improves the rate of resource utilization, but, at the same time, poses stronger demands on improving the memory efficiency and update performance of routing lookup. In response to this increasing requirements, some novel merging techniques [20–22] have been proposed. They first merge the tries built on all FIBs into one structure, and then compress it with uncontrollable update overhead, by sharing a larger number of trie nodes. However, only simple and original trie structures are involved in these approaches. Although many novel approaches with compact data structures [23–27] work well with single FIB, their scalability to virtual routers is limited. The key reason is the gap between their complicated updates and the aggregates of route updates from all FIBs.

In a word, although a large quantity of optimizations for the traditional model have been proposed, the applicability of most of them in large tables, IPv6, and some emerging techniques is limited. Actually, the essential limitation is the model itself.

1.2. Our approach

Instead of optimizing the traditional model, we propose a brand-new lookup model, *Split Routing Lookup Model* (*Split Model* in short). Our basic idea is to split the original FIB into two smaller ones, perform LPM on each of them, and then combine two results into the final result. During the “split” process, a great number of redundant information is produced by the partial similarity among prefixes. After eliminating redundancies, the on-chip structure is compressed significantly. Besides, after the “split” process, route updates are distributed to their targeted parts. In this case, update overhead is cut down. Meanwhile, the lookup performance is also improved by parallel processing on two new FIBs with shorter prefixes.

In this paper, we make the following key contributions:

1. We propose a new routing lookup model and prove its equivalence to the traditional model.
2. We present detail trie-based implementations of the proposed model.
3. We construct a comprehensive platform for performance evaluation. 6 trie-based approaches are implemented based on the traditional model and the proposed model respectively. Several metrics are measured, such as on/off-chip memory consumption, update overhead, worst-case lookup performance and so on. On this basis, the superiorities of our model is demonstrated in a comprehensive view.

The rest of this paper is organized as follows. Section 2 reviews related works that are implemented and evaluated in the experiments. Section 3 introduces Split Model with a proof of equivalent to the traditional model. Section 4 elaborates the trie-based implementation of Split Model with some core algorithms and the hash mechanism. The evaluation platform and experiment results are presented in Section 5. Then, Section 6 discusses Split Model’s application prospect, its limitation and our future works. At last, Section 7 concludes the whole paper.

2. Related work

2.1. Trie-based IP lookup

As shown in Fig. 2, the basic Trie is just a binary tree. Performing LPM needs to traverse it from the root according to each bit of the input address (0 for left and 1 for right). The traversing procedure terminates when a leaf is encountered or there is no proper path for next step. Along the traversing path, the longest prefix is treated as the matched result. To simplify the lookup process and to compress the node structure, a leaf-pushed trie (Fig. 3) is constructed by pushing all next hop information into leaf nodes. Even in this way, the worst-case memory accessing times for one lookup (32 for IPv4 and 64 for IPv6 [28]) is still undesirable, which is limited by the tree height.

2.2. Pipelined multi-bit trie

To reduce the tree height with total memory consumption controlled, a so-called Controlled Prefix Expansion (CPE) [13] technique was proposed for constructing multi-bit trie. Performing LMP is optimized by consuming two or more bits in each step (the number of bits consumed in one step is called a stride), reducing the worst-case lookup overhead as a result.

Furthermore, by mapping continuous tree levels onto independent SRAMs, a trie can be transformed into a multi-stage pipeline (Fig. 4). Since different SRAMs can be accessed simultaneously, a sequence of lookups are pipelined on such pipelined trie, enabling an average lookup speed as high as one lookup per cycle [17,18].

For efficient updates on the trie-based pipeline, Basu et al. [15] presented a so-called Write Bubble mechanism to map route updates into several *(stage, location, value)* triples. A group of such triples that target different stages are packed into a bubble, all memory writes produced by which can be performed in parallel. For the pipeline architecture, the maximal stage becomes the bottleneck of both memory consumption and update overhead. In
view of this, Basu et al. also proposed an efficient algorithm to construct multi-bit trie with the Maximal stage Minimized (MinMax).

2.3. Compressing techniques

By compressing a trie into a Directed Acyclic Graph (DAG) on basis of structural similarities, Shape Graph (SG) [24] achieves a significant lower memory-footprint. As shown in Fig. 5, its core idea is to label nodes according to their “shapes”, and combine all nodes with the same label. There are two basic labeling rules: (1) All leaf nodes are labeled as 1. (2) Two nodes have the same label if and only if their left child nodes have the same label while their right child nodes also have the same label.

Offset Encoded Trie (OET) compresses the leaf-pushed trie based on another labeling manner. After all non-leaf nodes are labeled in a top–down and left–right order, all non-leaf children of a node must have continuous labels (Fig. 6). So, for each node, an offset value, calculated by subtracting the label of its left-most child from its own label, can be used to locate all its children. Besides, each node also requires a Next-hop BitMap (NBM), of which each bit represents the type of its corresponding child (0 for a non-leaf child and 1 for a leaf node that must store some next hop information).

Both SG and OET compress the trie structure sharply, and thus enable better scalability to large routing tables with limited on-chip memory. As the cost, lookup on them can only answer “whether a matching can be found and how many bits are consumed when the matching happens?”, rather than “which next hop is corresponding to the input address?”. To answer the second question, the matched prefix, determined by the input address and the number of matched bits, is used as a key to retrieve the next hop information from an off-chip hash table.

2.4. Merging techniques for virtual routers

Considering a virtual router platform that consists of two FIBs shown in Figs. 1 and 7, two tries built on them respectively produce in total 23 nodes (Fig. 8). By overlapping these two tries with a lot of nodes shared, Trie Overlap (TO) [20] reduces the number of nodes from 23 to 17 (Fig. 9). To share more nodes between tries, Trie Braiding (TB) [21] is constructed by reversing some nodes of the second trie to produce more “similarities” between two tries before overlapping. In this way, 3 nodes are further eliminated (Fig. 10).

Besides, by pushing next hop information into leaf nodes, all non-leaf nodes in TO and TB are the same as those in the single trie. Furthermore, if the combined next hop information is stored in an off-chip table, leaving the index in trie structure, all leaf nodes also become the same as those in the single trie.

3. Split Routing Lookup Model

3.1. Motivation example

Our motivation is based on three fundamental observations about trie-based approaches:

2. Smaller upper bound of prefix length means less trie levels and also less trie nodes for the uni-bit trie. That’s why a uni-bit trie built on IPv4 prefixes always has less trie levels and trie nodes than built on the same number of IPv6 prefixes.

<table>
<thead>
<tr>
<th>Idx</th>
<th>Prefix</th>
<th>NextHop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Q1 0*</td>
<td>N1</td>
</tr>
<tr>
<td>1</td>
<td>Q2 1*</td>
<td>N2</td>
</tr>
<tr>
<td>2</td>
<td>Q3 11*</td>
<td>N2</td>
</tr>
<tr>
<td>3</td>
<td>Q4 010*</td>
<td>N3</td>
</tr>
<tr>
<td>4</td>
<td>Q5 100*</td>
<td>N2</td>
</tr>
<tr>
<td>5</td>
<td>Q6 1010*</td>
<td>N4</td>
</tr>
</tbody>
</table>

Fig. 7. The second FIB.
3. There are many structure similarities in a trie. That’s why Shape Graph \[24\] can compress the trie significantly. Moreover, such structure similarities essentially root in partial similarities among prefixes.

How to reduce trie levels? How to lower the upper bound of prefix length? And how to exploit the partial similarities? The key point of our answer is “splitting”. An example of the overview of our approach is shown in Fig. 11. In the transitional model (Fig. 11(b)), each input address is used to perform LPM on the FIB. For instance, given an address 10001*, its matched result \(P_6\) is found in the FIB directly.

But our approach is very different. For the basic idea, as shown in Fig. 11(a), all prefixes in the FIB are split into two shorter parts at a specified position (e.g. 10001 is split into 10 and 001 at position 2). Then, their front parts, as well as their back parts, make up two new FIBs respectively. In this way, the upper bound of prefix length in each new FIB is lowered. Due to partial similarities among prefixes, many redundancies are produced after splitting (e.g. 10001 and 11001 are similar at the back, so their back parts produced by splitting are just the same). Naturally, after integrating all redundancies, the size of each new FIB is reduced.

For lookup, the input address 10001 is required to be split at the same position (2). The produced two parts (10 and 001) are used to perform LPM on corresponding new FIBs respectively. Then, two lookup results (\(G_{14}\) and \(G_{22}\)) should be combined to form the final result. However, each prefix in new FIBs represents a set of prefixes in the original FIB (e.g. \(G_{22}\) represents both \(P_6\) and \(P_7\)). So, for proper combination, the intersection of two sets (\(P_5\), \(P_6\) and \(P_6\), \(P_7\)) is calculated. The final lookup result is just the longest prefix in that intersection (\(P_6\)).

To avoid intersection calculating, an advanced idea is proposed (shown in Fig. 11(c)). During the integrating process, it can be easily realized that all original prefixes, which are no longer than the split position (2), only appear in the new FIB composed of all front parts (\(FIB_1\) in Fig. 11(c)), and the rest original prefixes only appear in the new FIB composed of all back parts (\(FIB_2\) in Fig. 11(c)). In this way, the original FIB is essentially divided into two sub-sets according to the split position. All prefixes of the first one make up the first new FIB, while all back parts, produced by splitting all prefixes of the second one, make up the second new FIB.

For lookup, the input address 10001 is also required to be split at 2 into10 and 001. Then, 10 is looked up in the first new FIB, \(P_3\) is returned as a candidate result. Meanwhile, 001 is looked up in the second new FIB, \(G_{33}\), which represents \(P_5\), \(P_6\), is returned. Obviously, the \textbf{first part} of the input address (10) is also the first part of \(P_6\), but is not the first part of \(P_7\). So, only \(P_6\) is accepted as a candidate result. Among all candidates (\(P_3\) and \(P_6\)), the longest one \(P_6\) is selected as the final result.

3.2. Model assumption

**Memory Assumption.** In consideration of multi-path routing \[29\] and priority-based routing \[30\], the next hop information may be more complicated than a simple port. Consequently, it’s unwise...
to store complicated next hop information in the fast but expensive on-chip memory. So we assume that the lookup engine is deployed on chip, while the next hop table is stored in off-chip memory. Since on-chip memory is more scarce (in our experiment platform, it's the 32 M cache while the off-chip one is the 32 G memory), we pay our major attention to optimizing on-chip memory efficiency.

Update Assumption. Disruption of lookup by route updates affects the whole throughput, which gets worse in the virtual router platform. Accordingly, in this paper, a classic disrupt-free update mechanism, Write Bubble (as introduced in Section 2), is adopted in all related approaches. Consequently, the update overhead is simply measured as the number of produced write bubbles.

Lookup Assumption. For higher lookup speed, we assume that all trie-based lookup approaches are pipelined. In addition, according to the Memory Assumption, an off-chip memory access for next hop information is required to finish the lookup. Generally, it may become the throughput bottleneck [10]. So, we make it a principle that one lookup produce at most one off-chip memory access or multiple accesses that can be processed in parallel (we simply call this as one parallel off-chip memory access.).

3.3. Model definition

This subsection presents the definition of the basic Split Model, on basis of several fundamental definitions. An advanced model is discussed in the next subsection. All related notations are described in Table 1.

Definition 1. If $x$ and $y$ are two prefixes, then $x \leq y$ indicates that $x$ is a prefix to $y$.

Definition 2. Suppose $z$ is a set of prefixes, the longest one in it is denoted as $\text{Opt}(z)$.

Definition 3. (Traditional Model). Given the FIB $FIB$, for each input address $Addr$, routing lookup based on the traditional model is to seek for the next-hop information, of which the corresponding prefix in $FIB$ is determined through:

$$\text{prefix} = \text{Opt}(\text{Set})$$

where

Table 1

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FIB$</td>
<td>The original FIB, like the one shown in Fig. 11(b)</td>
</tr>
<tr>
<td>$Addr$</td>
<td>The destination address of an incoming packet</td>
</tr>
<tr>
<td>$Addr_1$, $Addr_2$</td>
<td>Two parts produced by splitting $Addr$ at $SPos$; $Addr_1$ is the front part</td>
</tr>
<tr>
<td>$SPos$</td>
<td>The Split Position</td>
</tr>
<tr>
<td>$FIB_1$, $FIB_2$</td>
<td>After splitting all prefixes of $FIB$ at $SPos$ into two parts and integrating redundancies, all front parts make up of $FIB_1$, and all back parts make up of $FIB_2$</td>
</tr>
<tr>
<td>$FIB_1$, $FIB_2$</td>
<td>$FIB_1$ is a subset of $FIB$ that consists of all prefixes that are no longer than $SPos$. After splitting rest prefixes of $FIB$ at $SPos$ and integrating redundancies, all back parts make up of $FIB_2$</td>
</tr>
<tr>
<td>$Set$</td>
<td>$Set$ is composed of all prefixes of $FIB$ that are prefixes to $Addr$</td>
</tr>
<tr>
<td>$Set_1$, $Set_2$</td>
<td>For all entries in $FIB$, that are prefixes to $Addr$, their corresponding original prefixes make up $Set_1$ (i.e., ${1, 2}$)</td>
</tr>
<tr>
<td>$Tree_1$, $Tree_2$</td>
<td>$Tree_1$ is the trie built on $FIB$, $i \in {1, 2}$</td>
</tr>
<tr>
<td>$HT_1$, $HT_2$</td>
<td>$HT_1$ is the hash table built on $FIB$, and cooperates with $Tree$, $i \in {1, 2}$</td>
</tr>
</tbody>
</table>

Definition 4. Suppose $p$ is a prefix has len bits in length, splitting it at position pos is just separating it from pos to generate two segments $p_1$ and $p_2$. Their lengths are $len$ and $len - pos$ respectively. What's more, such relationship is represented as $p = p_1 \oplus p_2$. And $p$ is called the original prefix of $p_1$ and $p_2$. Particularly, if $p$ is no longer than pos, the splitting process is defined as $p \Rightarrow p \oplus \phi$.

As mentioned in Section 3.1, there are three key points for the basic Split Model: (1) Splitting the original FIB into two new FIBs to lower the bound of prefix length. (2) Integrating redundancies produced after splitting to reduce the size of new FIBs. (3) Performing LMP on both new FIBs in parallel and combining lookup results to form the final result. Take the similar form as the definition of the transitional model, it can be defined as:

Definition 5. (basic Split Model). Given the FIB $FIB$ and the split position $SPos$. $FIB_1'$ and $FIB_2'$ are produced as their definitions in Table 1. For each input address $Addr = (Addr_1 \oplus Addr_2)$, routing lookup based on the basic Split Model is to seek for the next-hop information, of which the corresponding prefix in $FIB$ is determined through:

$$\text{prefix} = \text{Opt}(\text{Set}_1 \cap \text{Set}_2')$$

where

$$\text{Set}_1' = \{p \mid p \in FIB, p \Rightarrow p_1 \oplus p_2, \text{ and } p_1 \leq Addr_1\}, \ i \in \{1, 2\}$$

3.4. Advanced model

As mentioned in Section 3.1, the advanced Split Model aims to avoid the intersection calculating in the basic model. Actually, a significant effect is achieved by making a little change in making up two new FIBs.

Definition 6. (advanced Split Model). Given the FIB $FIB$ and the split position $SPos$. $FIB_1$ and $FIB_2$ are produced as their definitions in Table 1. For each input address $Addr = (Addr_1 \oplus Addr_2)$, routing lookup based on the advanced Split Model is to seek for the next-hop information, of which the corresponding prefix in $FIB$ is determined through:

$$\text{prefix} = \text{Opt}(\text{Set}_1 \cup \text{Set}_2) = \{\begin{cases} \text{Opt}(\text{Set}_1) & \text{Set}_2 = \emptyset \\ \text{Opt}(\text{Set}_2) & \text{Set}_2 \neq \emptyset \end{cases}$$

where

$$\text{Set}_1 = \{p \mid p \in FIB_1 \text{ and } p \leq Addr_1\}$$

and

$$\text{Set}_2 = \{p \mid p \in FIB, \ p \Rightarrow p_1 \oplus p_2, \text{ and } p_1 = Addr_1, p_2 \leq Addr_2\}$$

Actually, using $Addr_2$ to perform LPM on $FIB_2$, the returned result represents a group of original prefixes. The next step is to check whether $Addr_1$ is the first part of some of them. Thus, the procedure of calculating $\text{Set}_2$ can be transformed into:

$$\text{Set}_2 = \{p \mid p \in \text{Group}, \ p \Rightarrow p_1 \oplus p_2, \text{ and } p_1 = Addr_1\}$$

where

$$\text{Group} = \{p \mid p \in FIB, \ p \Rightarrow p_1 \oplus p_2, \text{ and } p_2 \leq Addr_2\}$$
Therefore, based on the advanced Split Model, the most time-consuming additional operation for lookup is to check whether Addr1 is the first part of some original prefixes, which are represented by the matched entry in FIB2. Furthermore, after a simple transformation discussed in next section, such operation can be easily carried out by efficient hash techniques [9, 10].

3.5. Model equivalence

This subsection presents the proof of that Split Model is equivalent to the transitional model.

Theorem 1. The basic Split Model is equivalent to the traditional model, namely

\[ Set = Set_1 \cap Set_2. \]

Proof. \( \forall \) prefix \( \in Set, \) according to Definition 3, prefix \( \in FIB \) and prefix \( \in Addr. \) Further, suppose prefix \( \Rightarrow prefix_1 \oplus prefix_2, \) we can get that prefix \( \in FIB \) and prefix \( \leq Addr, \) i.e. \( i \in \{1, 2\}. \) Then, according to Definition 5, prefix \( \in Set_1 \) and prefix \( \in Set_2, \) namely prefix \( \in Set_1 \cap Set_2. \) Thus,

\[ Set \subseteq Set_1 \cap Set_2 \quad (1) \]

\( \forall \) prefix \( \in \) SSet, then prefix \( \notin \) Addr \( \Rightarrow \) prefix, \( \not\in \) Addr1, or prefix \( \not\in \) Addr2, that's to say prefix, \( \not\in Set_1 \) or prefix \( \not\in Set_2. \) Accordingly, prefix \( \not\in Set_1 \cap Set_2, \) namely prefix \( \in \) S(Set \( _1 \cap Set_2). \) Thus,

\[ \overline{Set} \subseteq \overline{S}(Set_1 \cap Set_2) \quad (2) \]

Combining Eqs. (1) and (2), we can easily draw a conclusion that

\[ Set = Set_1 \cap Set_2. \quad \square \]

Theorem 2. The advanced Split Model is equivalent to the basic Split Model, namely

\[ Set_1 \cap Set_2 = Set_1 \cup Set_2 \]

Proof. \( \forall \) prefix \( \in Set_1 \cap Set_2, \) if prefix.length \( \leq \) SPos, then prefix \( \in Set_1. \) Otherwise, prefix.length > SPos, suppose prefix \( \Rightarrow prefix_1 \oplus prefix_2, \) then prefix \( \leq Addr, \Rightarrow \) prefix = Addr. Besides, prefix \( \in Set_1 \Rightarrow \) prefix \( \in Set_1, \) according to Definition 6, we can get that prefix \( \in Set_1. \) Therefore, \( \forall \) prefix \( \in Set_1 \cup Set_2, \)

\[ Set_1 \cap Set_2 \subseteq Set_1 \cup Set_2 \quad (3) \]

\( \forall \) prefix \( \in Set_1, \) according to Definitions 1 and 5, we can get that prefix \( \in Set_1 \cap Set_2. \) According to Definition 6, it's obvious that prefix \( \in Set_2, \) \( \Rightarrow \) \( prefix = Addr_1. \) Moreover, prefix \( \not\in Addr_1 \Rightarrow prefix \in Set_1. \) Thus, \( \forall \) prefix \( \in Set_1 \cap Set_2, \)

That's to say, \( \forall \) prefix \( \in Set_1 \cup Set_2 \Rightarrow \) \( prefix \in Set_1 \cap Set_2, \) Namely,

\[ Set_1 \cup Set_2 \subseteq Set_1 \cap Set_2 \quad (4) \]

Combining Eqs. (3) and (4), we can easily draw a conclusion that

\[ Set_1 \cap Set_2 = Set_1 \cup Set_2. \quad \square \]

In a word, according to Theorems 1 and 2, it can be deduced that both the basic and the advanced Split Model are equivalent to the traditional model. But the basic Split Model is not so convenient for implementation, because it requires a time-consuming operation, intersection calculating, to finish the lookup. So, only the implementations of the advanced model is discussed in the next section. Meanwhile, in the rest of this paper, Split Model is used to refer to the advanced Split Model.

4. Trie-based model implementation

4.1. Uni-bit Split Trie

On basis of Split Model, the original FIB is split into FIB1 and FIB2 at SPos. For each input address Addr, it is also split at SPos first, producing two shorter ones (Addr1 and Addr2). They are then looked up in FIB1 and FIB2, respectively. At last, their lookup results should be combined. So, to implement Split Model, three issues need to be addressed: (1) How to look up Addr1 in FIB1, (2) How to look up Addr2 in FIB2, (3) How to combine lookup results.

According to the definition of Split Model (see Definition 6 and Table 1), FIB1 is just a subset of FIB, and the result of looking up Addr1 in it (say Opt(Set1)) is just the LPM result. So, a uni-bit trie built on FIB1 can be adopted to perform lookup of Addr1 in traditional way.

But the lookup of Addr2 in FIB2 can only find out a group of original prefixes, from which to pick out the truly LPM result also requires a checking using Addr2, as the second step. Anyway, a uni-bit built on FIB2 is also useful for the first step to find out all candidates.

According to Definition 6, the result combination can be done in a simple way. For LPM, the result of looking up Addr2 in FIB2 has a higher priority. In another word, it will be accepted as the final result as long as it's a valid result. Otherwise, the result of performing LPM in FIB2 using Addr2 is accepted as the final result.

As shown in Fig. 12, in addition to two uni-bit tries (Tree1 and Tree2) built on FIB1 and FIB2 respectively, a virtual root node is also required to store the split position and two tree pointers. These components make up a uni-bit Split Trie. The lookup approach for it is shown by the following examples.

To look up 10001, it's first split at 2 into two parts, Addr1 (10) and Addr2 (001), which are issued to Tree1 and Tree2 respectively for parallel processing. All matched entries of Addr2 on Tree2 are \( G_i(P_i), G_i(P_{i2}) \) (and \( P_i \)). By checking Addr2 (10) toward all candidate original prefixes \( P_i, P_{i2} \), only \( P_2(1000) \) and \( P_2(001) \) are matched. So, the longer one \( P_2(1000) \) is returned as the lookup result on Tree2. Moreover, it is directly accepted as the final result, even though the lookup of Addr1 (10) on Tree1 also succeeds (\( P_i \) is returned as the result). However, to look up 01001, the result of looking up Addr2 (01) on Tree2 is accepted as the final output, because no one of the candidates, which are found out by looking up Addr2 (001) on Tree2, starts by 01.

4.2. Off-chip table design

As mentioned in Section 3.2, to keep scarce on-chip memory away from complicated next-hop information, the next-hop table is stored off chip. As a result, for each lookup, an off-chip memory access is necessary.
For Split Model, the off-chip table is also “split”. More specifically, since the lookup in Tree$_2$ is just the same as traditional LPM, all next-hop information of FIB$_1$ are organized as a direct-access table (HT$_1$), of which each entry can be directly accessed by the index of its corresponding prefix in FIB$_1$ (the direct access can be also treated as a special hashing).

However, performing LPM on Tree$_2$ using Addr$_2$ can only find out a group of candidate original prefixes. To pick out the final result, their first SPos bits are checked in comparison to Addr$_1$. Accordingly, only the first SPos bits of each entry in that group are used for picking final result. Thus, by making up the candidate group by the first SPos bits of all candidate original prefixes, what’s required to pick out the final result is now a simple member checking to a set. It can be easily solved with fast speed by all classic hashing mechanisms.

As shown in Fig. 13, each entry of the hash table (HT$_2$) for Tree$_2$ consists of two fields: the key represents the first SPos bits of the original prefix corresponding to this entry, and the value denotes the next-hop information. Let’s take an example to show how these off-chip tables (HT$_1$ and HT$_2$) work for lookup in Split Trie.

To look up 10001, it’s two parts Addr$_1$ (10) and Addr$_2$ (001) are looked up in Tree$_1$ and Tree$_2$ respectively. An LPM result P$_1$ is returned when looking up Addr$_1$ (10) in Tree$_1$, and it’s used as an index to fetch the next-hop information (N$_1$) from the direct access table HT$_1$. On the other hand, during the lookup of Addr$_2$ (001) in Tree$_2$, three groups (G$_0$, G$_1$ and G$_2$) are encountered. Each of them corresponds to a group of entries in HT$_2$. And it’s required to perform hashing to these groups using Addr$_2$ (001) as a key, in the encountering order of their corresponding indexes. Although the hashing for both G$_0$ and G$_2$ are successful, the hit entry returned for G$_2$ (10 N$_0$) has a higher priority because G$_0$ is encountered later and thus represents a longer original prefix. Then, N$_0$ is returned as the lookup result in Tree$_2$. And it’s accepted as the final lookup result at last.

Since many excellent hashing techniques have been proposed to provide compact structures with low collision rate, we are not going to discuss a new technique here. In our scheme, a widely used multiple hash technique [9] is implemented to organize HT$_2$. Besides, by using multi-port memory or multiple parallel memory modules, multiple off-chip memory accesses required by hashing can be processed in parallel.

4.3. Leaf pushing

Due to the non-leaf-pushed structure, hashing to a candidate group has to be performed many times during the traversing procedure on Tree$_2$. This may result in accessing off-chip memory many times and break our principle about off-chip access (see Section 3.2).

To address this issue, Leaf-Pushing should be performed. For Tree$_1$, this can be done simply, just using the traditional leaf-pushing algorithm [13]. But for Tree$_2$, leaf-pushing should be performed very carefully. Actually, what are stored in Tree$_2$ are indexes that can be used to locate groups of hash entries in HT$_2$. And each of those groups represents a group of original prefixes in fact. So, pushing an index (the pushed index) from one node to another node that also hold an index (the object index) is actually pushing a group (the pushed group) of original prefixes to another group (the object group). For information integration, simply discarding the pushed index, like traditional leaf-pushing algorithm, is not available. That’s because there may be no “group-level” prefix relationship between the pushed group and the object group. For instance, when pushing G$_1$ (P$_1$(0000)) to G$_2$ (P$_1$(10001)), G$_1$ can not be discarded since P$_1$(0000) is not a prefix to either P$_1$(100001) or P$_1$(1100). In contrast, it’s ok to discard G$_2$ when pushing it to G$_3$.

Therefore, to perform leaf-pushing properly, a new algorithm for integrating information during the pushing process is designed. When pushing an index to another in Tree$_2$, their corresponding groups of prefixes should be merged into one group, then the pushed index is discarded. For two groups, they are merged in this way: inserting all prefixes of the pushed group to the object group except that are prefixes to some entries of the object group. For example, when pushing G$_1$ to G$_3$, G$_1$ is discarded after G$_3$ has received a new entry P$_4$.

To keep our system work, this process must be reflected on the hash table HT$_2$. The key point is to make it available that the relationship between two original prefixes can be judged by their corresponding entries in HT$_2$. Considering that each entry in HT$_2$ is formulated as (key, value). If the keys of two entries are different, it means their corresponding prefixes differ at the front part. In another word, there is no prefix relationship between them. But if their keys are equal, how can the correct judgment be made? Actually, for two prefixes corresponding to the pushed index and the object index respectively, their back parts must along the same traversing path on Tree$_2$. If their first parts (represented by key filed of their corresponding entries in HT$_2$) are the same, there must be a prefix relationship between them. But which is the prefix is also indefinite.

So, a length information is also required. For LPM, only the entry corresponding to the longer one of those two prefixes should be served in HT$_2$. So, by adding a new filed len, which represents the length of the prefix corresponding to this entry, to every entry in HT$_2$, all problems are solved. The proposed leaf-pushing algorithm, named Information Integrating and Pushing (IIP in short), is described in pseudo code in Algorithm 2.

---

Algorithm 1. Merging (cur_idx, push_idx) /* merging two group of entries in HT$^2$ */

Input: cur_idx, push_idx /* push push_idx to cur_idx. */

Output: new_idx /* return the new index after merging */

if cur_idx = DEFAULT then
  | "If no entries corresponding to the current index, return the pushed index directly." |
  return push_idx;

push_group = HT$^2$(cur_idx).group;
push_group = HT$^2$(push_idx).group;
for i = 0 to push_group.elen.num - 1 do
  if hit.elen = cur_group.hash[push_group.elen[i].key] then
    hit.elen = push_group.elen[i].elen;
    hit.elen.value = push_group.elen[i].value;
  end
end

return cur_idx;

---
Algorithm 2. InflntPush(node, idx)/* Push index down to leaf nodes */

Input: node, idx/* The pushed index idx has arrived in current node node */

idx = merging (node, idx, idx);
for i = 0 to node.child.num - 1 do
  if node.child[i].ptr != NULL then
    InflntPush (node.child[i].ptr, idx); /* Push new idx to its child
  end
end

4.4. Multi-bit Split Trie

Besides Leaf-Pushing, Multi-bit Trie is another widely used technique for trie-based approaches. It can also be implemented in Split Model in a straightforward way. Take the Controlled Prefix Expansion [13] as an example. Given a uni-bit Split Trie (such as the one shown in Fig. 12), and the object trie level, say level, a multi-bit Split Trie can be constructed in the following way. For two sub trees of this uni-bit Split Trie, one of them is selected to perform CPE with an objective level equals to level, and the other is used to perform CPE with an objective level no longer than level. During these processes, Algorithm 2 is also used for information integration. In this way, a multi-bit Split Trie with level levels is constructed (as shown in Fig. 15). Actually, there are many choices to select sub trees and to assign objective levels. Among these choices, the one, using which the constructed multi-bit Split Trie has the minimized memory size, is selected.

4.5. Update in Split Trie

Since the original FIB has been split, route updates are also diverged. More specifically, all updates toward the prefixes that are no longer than SPos only reflect to Tree1 and HT1. While the rest updates will only affect Tree2 and HT2. As a result, additional update overhead, such as that resulted by leaf-pushing, are reduced. For instance, to delete P3(10000+), it’s only required to delete its corresponding entry in HT2 (Fig. 16(c)).

5. Experimental evaluation

5.1. Evaluation platform

To evaluate Split Model, we selected and implemented 6 trie-based approaches. The first two are classic techniques for constructing efficient multi-bit trie: Controlled Prefix Expansion (CPE) [13] and MinMax (MM) [15]. The next two are novel compressing techniques: Shape Graph (SG) [24] and Offset Encoded Trie (OET) [25]. The rest two are recent merging techniques for virtual routers: Trie Overlap (TO) [20] and Trie Braiding (TB) [21]. These approaches are all implemented and evaluated on basis of both the traditional model and the proposed Split Model. Their implementations on Split Model are named as Controlled Split Prefix Expansion (CSPRE), Split MinMax (SMM), Split Shape Graph (SSG), Offset Encoded Split Trie (OEST), Split Trie Overlap (STO) and Split Trie Braiding (STB) respectively.

Besides, 14 real-world, public Border Gateway Protocol (BGP) routing data are collected from [31] for evaluation. Each of them consists of an IPv4 FIB, an IPv6 FIB and a whole day’s update traces (detail characteristics are depicted in Table 2). Based on these data, we first conduct experiments for single FIB, and then turn to the virtual router platform (composed of 14 routers). All experiments are run on a DELL T620 server with an Intel Xeon E5-2630 CPU (2.30 GHz, 6 Cores) with 32 GB memory.

For memory efficiency, both on-chip and off-chip memory consumption are measured. The update overhead is measured as the number of write bubbles produced by a whole day’s route updates. Besides, the pipeline depth, which represents the lookup latency and the worst-case memory accesses for one lookup, is measured for an overview of lookup performance.

5.2. Performance for single FIB

5.2.1. Key model parameter and performance overview

The Split Position is the key parameter for Split Model, because it determines the structure shape and thus affects the performance. Particularly, when it equals to 0 or the maximum of prefix length (32 for IPv4 and 64 for IPv6), the Split Model turns back to the traditional model. In this view, the traditional model is just a special case of Split Model.

Figs. 17 and 18 show curves of the on-chip memory consumption and the update overhead varying with the split position for
4 approaches, which are all constructed as 8-level pipelines based on rrc00’s IPv4/6 FIBs. As depicted, no matter for IPv4 or IPv6, no matter for memory efficiency or update overhead, and no matter for which approach, the curve is concave and the valley value is reached around the middle.

For the IPv4 FIB, a multi-bit trie (the split position equals to 0 or 32) constructed by CPE, which aims to minimize the memory consumption of multi-bit trie, consumes 3 MB on-chip memory. While the on-chip memory footprint of a MST constructed by CSPE is only 22 KB, if 15 is selected as the split position. Due to compact on-chip structures, SG and OET compress the on-chip memory size below 800 KB, which can still be reduced by an order of magnitude based on Split Model. Actually, for these 4 approaches, the Split Model reduces on-chip memory consumption and update overhead by 82.4–99.2% and 50.3–98.7% respectively, when the split position varies from 12 to 20. Those reductions for the IPv6 FIB are 14.9–94.8% and 29.0–90.7% respectively. It seems, from these results, that Split Model is not so useful for IPv6 as for IPv4. The key reason is that today’s real-world IPv6 FIBs are too small to reflect its superiorities.

On the other hand, besides that Split Model can compress the on-chip memory by one or two orders of magnitude, all implemented approaches on it keep their own characteristics. For instance, although both CPE and MinMax optimize the multi-bit trie by selecting proper strides, their objectives are very different. CPE aims to minimize the total memory consumption, while MinMax is used to reduce the update overhead for trie-based pipeline by minimizing the maximal stage. Consequently, for their “split” versions, CSPE always has better memory efficiency than SMM, while the situation for update overhead is just the opposite.

For another example, SG compresses the on-chip memory by transforming the trie structure into a DAG on basis of the structure similarity. It essentially reduces a great number of trie nodes, but the node structure is not optimized and even becomes more complicated due to additional bitmaps [24]. Consequently, SG is more useful in IPv4. Because the original trie built on an IPv4 FIB always has much more nodes than that on an IPv6 FIB (real-world FIB nowadays). Another reason is that shorter prefixes in IPv4 FIBs re-
sult in smaller strides, which in fact leads to “lighter” nodes. By contrast, OET has much better performance in IPv6, for it gains significant effect on compressing node structure without any change on the number of nodes. For SSG and OEST, such difference is also clearly demonstrated in Figs. 17 and 18.

More details about how SSG and OEST (the selected split positions are 15 and 30 respectively) work on rrc00’s IPv4/6 FIBs with different strides are shown in Figs. 19 and 20 respectively. As depicted, their memory foot-prints increase, with some ups and downs, as the stride is increasing. Especially for the IPv6 FIB, SSG’s on-chip memory consumption, as well as its update overhead, even grow exponentially after the stride exceeds 8. From Figs. 19 and 20, we can also see that the 2-stride SSG has the best memory efficiency and update performance. However, its small stride will lead to long lookup latency. By contrast, the 6-stride OEST achieves the best update performance, while keeping desirable memory efficiency and trie level. Since on-chip memory has been sharply compressed for both SSG and OEST, the most important issue turns to update overhead. In this view, the 6-stride OEST (split at 15 and 30 for IPv4/6 respectively) has a desirable tradeoff between memory efficiency and update overhead.

5.2.2. Memory consumption and update overhead over FIBs

Fig. 21 shows the on-chip/off-chip memory consumption and a whole day’s update overhead of the 6-stride and OET and the 6-stride OEST (split at 15) built on 14 IPv4 FIBs. For each of these FIBs, OET has compressed on-chip memory to almost 512 KB. But it’s not enough, OEST further reduces this consumption by an order of magnitude. More specifically, 32 KB on-chip memory is enough for the 6-stride OEST to hold any of these FIBs. As a matter of fact, in comparison with OET, OEST gains an on-chip memory reduction by a factor of 92.1–97.5% in all cases, while its update overhead is also reduced by 95.8–99.5%.

However, to ensure the off-chip hashing be finished in at most one parallel memory accessing, a large amount of off-chip memory are required. As shown in Fig. 21, OEST’s off-chip memory usage increases to one or two orders of magnitude larger than OET. In the best case (rrc06), OEST consumes 80.3 MB off-chip memory, which is 9.8 times as OET’s off-chip memory usage (2.5 MB). While in the worst case (rrc03), OEST requires in total 273.3 MB off-chip memory, which increases by almost 105 times in comparison to OET’s 2.6 MB off-chip memory consumption. Even in this case, the exchange from off-chip memory to on-chip memory in Split Model is still worthwhile, not to mention its significant improvement on update performance.

5.2.3. Worst-case lookup overhead

According to our model assumption (see Section 3.2), a trie-based structure is usually mapped onto a multi-stage pipeline to achieve an average throughput as high as one lookup per cycle. In this way, the lookup performance is determined by the pipeline depth, which represents not only the lookup latency but also the worst-case memory accesses for one lookup.

In Split Model, the original trie is split into two shorter subtrees. Lookups on them are independently and thus can be processed in parallel. Moreover, by mapping them onto the pipeline side by side, the pipeline depth is cut down. We measured the pipeline depth of OEST with different strides and different split positions (if the split position equals 0, it’s just OET in fact). Table 3 shows the results.

It’s clearly demonstrated that OEST produces shallower pipelines in all cases. In the best case, the pipeline depth is cut down to half by OEST. Even in the worst case, such a reduction is still 25.0%. It’s also noteworthy that the pipeline depth varies in a “symmetry” way with the increasing of the split position if the stride is fixed. More specifically, with two split positions that have the same absolute value of difference from the median, the produced two pipelines must have the same depth. But, as shown in Figs. 17 and 18, the curves of memory efficiency and update overhead are not so symmetry. Actually, around the median of all split positions, the smaller one between two symmetry split positions always leads to better performance on both memory and update. Therefore, the “best” split position must appear before the median, in consideration of on-chip memory foot-print, update overhead and worst-case lookup overhead together.

5.3. Performance for virtual routers

To evaluate the scalability of Split Model to virtual routers, a virtual router platform is constructed with 14 real-world router instances. Based on this platform, STO and STB are evaluated with different split positions. Figs. 22 and 23 show the similar scenario as that for single FIB: the curves of on-chip memory and update overhead are all concave, and their valley values are achieved around the middle.

On the other hand, since TB aims to enlarge the quantity of sharing nodes between all member tries, it outperforms TO on both memory efficiency and update overhead in most cases. This superiority for TB is reserved in Split Model. Take the situation in IPv4 for an example. If the split position varies from 12 to 20 (as shown in Fig. 22), SSTB, in companion to SSTO, reduces on-chip memory foot-print and total number of produced write bubbles by a factor
of 3.5–28.9% and 18.9–33.5% respectively. While such reductions for TB (in comparison to TO) are only 3.1% and 8.0% respectively. That’s to say, TB’s superiority are boosted in Split Model.

Actually, by utilizing STB, 160 KB on-chip memory is enough to store all 14 IPv4 FIBs together if 15 is selected as the split position. Considering that these FIBs have in total 6,495,433 prefixes (see Table 2), STB requires 11.4 KB for each FIB and 0.2 bits for each prefix on average. With the same split position, only 13,064 write bubbles are produced by the aggregates of a whole day’s updates from all FIBs. For IPv6, with a split position of 30, STB consumes in total 580.0 KB on-chip memory to store all 14 FIBs, and produces 41,365 write bubbles for the aggregates of updates. These results demonstrate clearly the scalability of Split Model to virtual routers.

### 6. Discussions and future works

#### 6.1. Model applications

Recently, the GPU has been proved to be of value for high-speed software routers [32,33]. Many approaches have been proposed to deploy efficient routing lookup engines on the GPU platform [34,35]. However, due to the data-parallel manner, to benefit more power from the massive parallelism of the GPU, a large number of packets should be processed on it at a time. In this way, the lookup throughput will be pushed to a high level at the cost of extra look-up latencies in waiting for too many packets. If the GPU’s parallelism can be well utilized with less packets, such a throughput-latency dilemma will be resolved essentially. Split Model provides a good application prospect in this view. After splitting the original FIB into smaller ones, the lookup overhead on each of them is reduced by nearly half. Most importantly, since these new FIBs are completely independent, lookup on them can be processed in parallel. Thus, by combining the data-parallel manner and a no-communication task-parallel manner, the GPU’s parallelism will be well benefited with less packets. The rest problems are how many parts should the original FIB be split into, when and how to perform results combination. These are our underway works.

In addition to IPv4/6 addressing, there are many other transport techniques and protocols widely used nowadays, such as Multicast (Anycast addressing and MultiProtocol Label Switching (MPLS) [36,37]. Based on group communications, Multicast has significant effects on saving bandwidth, especially in Data Center Networks (DCN) [38]. As an overlay technology, MPLS provides efficient forwarding between Label-Switched Routers (LSR), by adding a new label to the header of a frame. Besides, it has desirable flexible to satisfy many Quality of Service (QoS)-related requirements. Although many common approaches have been proposed to solve multicast addressing (for anycast addressing, the solution is similar) and MPLS label lookup, merging them into a unified prefix tree with IP also has many benefits [23]. But the cost is the upper bound of prefix length may be double. In this view, Split Model must be more useful than the traditional lookup model, for it can cut down prefixes while achieving improvements on the comprehensive performance.

#### 6.2. Model limitations

The large off-chip memory usage must be the key limitation of Split Model. In exchange for high on-chip memory efficiency, fast incremental updates and low lookup overhead, additional off-chip memory are required in Split Model. An off-chip hashing is neces-
sary to finish each lookup. To confine the hash process to being done in at most one parallel memory access (multiple accesses that are processed in parallel), the hash table is kept very sparse to avoid hash conflicts as possible. As a result, the off-chip memory efficiency is lost. Although, according to our model assumption, this exchange is worthwhile today, it indeed restricts the Split Model to satisfy some future requirements. Such as for future big IPv6 FIBs, or for a virtual router platform that holds thousands of router instances. Therefore, to improve the scalability of Split Model, this limitation must be broken.

On the other hand, there are indeed some spaces for improving the hashing. In our case, the hash table has two obvious characteristics: (1) The key for hashing is of fixed-length (equals to the split position), (2) The hash table is naturally grouped. Unfortunately, they are not benefited enough by classic hash mechanisms, such as the one used in this paper. Therefore, our future work is to study on new hash mechanisms that can benefit more from these characteristics of Split Model to improve off-chip memory efficiency with fast hash speed.

7. Conclusion

In this paper, we have proposed a novel routing lookup model, Split Model, with trie-based implementations, and then proved its equivalence to the traditional model. With 14 real-world routing data, a comprehensive evaluation platform was constructed by implementing 6 efficient trie-based approaches on both traditional model and the proposed model. Based on this platform, a series of experiments were conducted. The experimental results demonstrated clearly the superiorities of Split Model in on-chip memory efficiency, update overhead, and pipeline depth. Besides the performance for single FIB, we also evaluated Split Model on the virtual router platform with in total 14 real-world router instances. The results are also encouraging. All these improvements are achieved at the cost of a large off-chip memory footprint. But such an exchange is worthwhile since we always have enough off-chip memory, in comparison to the consumption of today’s applications. Anyway, this issue, as one of our future works, should also be addressed to satisfy future requirements.

Acknowledgments

This work is supported by the National Basic Research Program of China (973) under Grant 2012CB315805, and the National Science Foundation of China under Grant 61173167.

References