Learned FIB: Fast IP Forwarding without Longest Prefix Matching

Shunsuke Higuchi*, Yuki Koizumi*, Junji Takemasa*, Atsushi Tagami† and Toru Hasegawa*
*Graduate School of Information Science and Technology, Osaka University
†KDDI Research, Inc.

***Abstract*—This paper proposes an IP forwarding information base (FIB) encoding leveraging an emerging data structure called a *learned index*, which uses machine learning to associate key-position pairs in a key-value store. A learned index for FIB lookups is expected to yield a more compact representation and faster lookups compared to existing FIBs based on tries or hash tables, at the cost of efficient FIB updates, which is difficult to support with a learned index. We optimize our implementation for lookup speed, exploiting that for efficient FIB lookups it is enough to approximate the key-position pairs with a piece-wise linear function, instead of having to learn the key-position pairs. The experiments using real BGP routing information snapshots suggest that the size of the proposed FIB is compact and lookup speed is sufficiently fast regardless of the length of matched prefixes.**

***Index Terms*—Forwarding, learned index structure, longest prefix matching**

I. Introduction

Packet forwarding is one of the essential functions of IP, and therefore much effort has been devoted to fast IP forwarding [1]–[3]. IP forwarding has suffered from the complex table lookup operation, i.e., longest prefix matching, due to either the exploitation of ternary content addressable memory (TCAM) or the adoption of classless inter-domain routing (CIDR). That is, IP routers have to search their forwarding information bases (FIBs) for the longest IP prefix that matches the destination IP addresses. IP addresses and IP prefixes are referred to as addresses and prefixes for notation simplicity. This study aims at realizing fast FIB lookup by avoiding longest prefix matching with an emerging data structure.

Longest prefix matching for IP forwarding is considered an already solved issue; however, the increase in the number of prefixes due to the trends in Internet business makes us revisit fast longest prefix matching. One trend is that Internet service providers (ISPs) are increasingly using longer prefixes than before to control customer's traffic finely. The other is that IPv6 is being deployed and thus in the near future the number of IPv6 prefixes will become much larger than that of IPv4 prefixes.

To circumvent the high energy consumption of TCAM devices, two types of FIBs for longest prefix matching, which are assumed to use only SRAM devices, were developed: Trie- and hash-based FIBs. They address a problem raised by CIDR that an address matches multiple prefixes since CIDR enables to aggregate prefixes. Longest prefix matching rather than exact matching is used to search for the longest matched prefix.

Longest prefix matching is performing well in the current Internet. It is, however, expected that trie- and hash-based FIBs cannot scale to the increase in the number of IP prefixes in terms of the computation time. The computation time of longest prefix matching based on trie-based FIBs is proportional to the average prefix length [4]. Although level compression techniques, such as LC-trie [5] and Poptrie [6], reduce the apparent length, such compression is not effective for long prefixes. Similarly, the computation time of hash-based FIBs increases in proportion to the average prefix length.

In this paper, we adopt exact matching rather than longest prefix matching, inspired by the emergence of an index structure for key-value stores (KVSs), referred to as a *learned index structure* [7]. We refer to a learned index structure as a *learned index* for simplicity. A learned index uses machine learning to associate keys and their positions in the KVS. The core idea to enable IP forwarding with exact matching is that FIB entries are divided into several sub entries so that all the prefixes in the sub entries are disjoint. The idea is similar to the leaf-pushing technique for tries [8].

The paper is inspired by the two studies: One study proposes to use a learned index for IPv4 prefix matching. Higuchi et al. [9] design an FIB lookup method according to exact matching by leveraging a learned index and preliminarily demonstrate that a learned index enables to construct a compact FIB with fast lookup operation, which requires half the memory size compared to the FIB based on LC-tries [5] and realizes the same lookup speed. Another study, which does not directly relate to longest prefix matching, reveals an important insight into a learned index. Rashelbach et al. [10] reveal that a learned index is a piece-wise linear function, provided that all the machine learning models in the learned index are neural networks with fully connected layers and the ReLU activation function. They utilize this feature to derive the worst-case error analytically and to realize efficient learning of models.

We argue that the essential feature of the learned index is not to use machine learning to search a KVS but to approximate the key-position relation of the KVS with a combination of piece-wise linear functions, each of which is expressed by a small neural network. Unlike the aforementioned two studies [9], [10], which rely on machine learning techniques for constructing a learned index, we propose a deterministic method to directly design weights and biases of neural net-

This work has been supported by KAKENHI Grant-in-Aid for challenging exploratory research 21K19771.

978-1-6654-4131-5/21/$31.00 ©2021 IEEE

works in a learned index.

The proposed method can design neural networks in a learned index so that the maximum prediction error of the neural networks is lower than the pre-determined threshold. The proposal enables to optimize a learned index in the following three ways: First, the proposal minimizes the number of layers in neural networks of a learned index. Specifically, we use neural networks of a single hidden layer because we do not need to increase the number of layers to reduce the prediction error. Second, the proposal eliminates multiple iterations of the neural network computation, whereas the original learned index recursively applies neural network computation to achieve better predictions. This is because we can control the maximum prediction error without applying multiple neural networks. Third, it optimizes the implementation of a learned index to utilize the capability of state-of-the-art CPUs. Specifically, with the proposal, the lookup program for a learned index is entirely realized without loops and branches, thereby achieving fast lookup.

In this paper, we propose an FIB based on the aforementioned optimized learned index. We refer to an FIB based on a learned index as a *learned FIB*. The contributions of this paper are summarized as follows:

- We propose a method to apply a learned index to encode an FIB, which results in a more compact representation than existing trie-based encodings [6]. This requires, however, eliminating longest prefix matching since learned indexes only support exact matching. This sacrifices efficient FIB updates. Correspondingly, the current learned FIB mainly focuses on encoding static FIBs, which change only infrequently.
- We utilize the essence of learned indexes, which is to approximate the key-position relations with piece-wise linear functions. Based on this fact, we develop an algorithm to design a learned index nearly optimal in terms of its implementation. The proposed method is applicable to not only IP FIBs but also general databases, which are the original target of learned indexes.
- We develop an implementation of the learned FIB optimized for recent CPUs. Using the implementation, we demonstrate the following three factors: First, the FIB lookup speed of the learned FIB is independent of the prefix length. Second, the FIB lookup speed is sufficiently fast though it is slightly lower than Poptrie [6], which is one of state-of-the-art FIBs. Finally, the learned FIB is compact and it fits into the second-level cache of a recent CPU.

The rest of this paper is organized as follows: We introduce the related work in Section II while describing the concept of learned indexes in Section III. We describe the design rationale of the learned FIB in Section IV, design the learned FIB in Section V, and implement the learned FIB in Section VI. We discuss open issues in Section VII. We evaluate the performance of the implemented learned FIB in Section VIII. Finally, we conclude this paper in Section IX.

II. Related Work

Fast IP forwarding is a traditional and fundamental research topic, and there have been many solutions based on both hardware and software. TCAM is widely used by hardware-based solutions [11]. TCAM realizes constant-time complexity by parallelizing matching operations for all stored prefixes. Another hardware-based solution is to leverage FPGA to parallelize search operations of multiple hash tables, each of which maintains a different length of prefixes [12]. However, the size of TCAM and FPGA's on-chip memory is limited, and thus an FIB does not entirely fit into such memory device as the number of prefixes grows [13].

With the evolution of general-purpose CPUs, software-based solutions have been revisited. The key for high-speed FIB lookup is to fit an FIB into a fast but small memory device, such as CPU caches, and thus compact trie-based data structures have been studied well. Tries, however, reduce space complexity at the cost of time complexity, and thus existing studies aimed at reducing time complexity. Traditional effort [5], [14]–[16] has reduced the number of vertices traversed during longest prefix matching without sacrificing space complexity much. A state-of-the-art trie-based solution, Poptrie [6], employs two techniques: a multi-bit trie [16], where multiple bits of the prefix are matched in traversing each vertex, and direct pointing [15], [17], which skips matching of upper bits of a prefix by a single access to an array that stores pointers to sub trees of the trie. Asai and Ohara [6] have reduced the number of instructions required to traverse each vertex by leveraging an AVX instruction in addition to the two techniques.

Despite the numerous effort, the lookup speed of the state-of-the-art trie-based solutions [6] still depends on the length of matched prefixes because they must traverse vertices depending on the length. Prefixes equal to or longer than 24 bits dominate the current Internet traffic [18], and the length of prefixes will be longer in the future due to the IPv6 deployment. Hence, our approach, learned FIB, is designed to keep constant-time complexity regardless of the prefix length.

III. Learned Index

Learned index structures use one or multiple machine learning models to associate keys and the positions of the corresponding entries in a KVS. For notation simplicity, we refer to a machine learning model and a learned index structure as a model and a learned index, respectively. An overview of a learned index is illustrated in Fig. 1. The algorithm of learning indexes consists of the training and the search phase since learned indexes rely on machine learning.

In the training phase, entries, pairs of a key and a value, are sorted in ascending order of the keys and stored in an array. An example of such an array is illustrated in Fig. 3 (the table in the left-hand side of the figure). The mapping from the keys to the positions is regarded as a monotonically increasing function, and therefore a regression model learns the key-position mapping. We refer to the mapping from the keys to the positions as the *target function*.

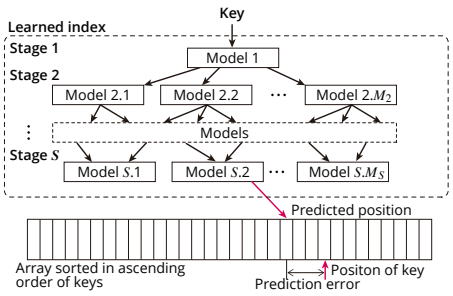


Fig. 1. An example structure of learned index (recursive model index)

The search phase consists of the prediction and the local search phase. First, given a key, the position of the key is predicted with the models in the prediction phase. Unlike traditional index structures, such as B-trees and hash tables, the position predicted by the models contains a certain degree of prediction error. The prediction error is defined as the difference between the predicted and the actual position of the key. Thus, a local search around the predicted position is performed to find the key in the local search phase.

Accurate prediction is indispensable to minimize the speed of a local search operation. Kraska et al. [7] propose a *recursive model index (RMI)*, which uses multiple small models hierarchically, as shown in Fig. 1, rather than using a single big model. Given a key, at each stage, one of the models takes the key as an input and generates the prediction result. The model in the next stage is selected according to the prediction result, and the selected model generates the prediction result recursively. Finally, the prediction result of the last stage is used as the output of the RMI. Since each model is responsible for a smaller area of the key-space than the case of a single big model, the RMI predicts the positions of keys with lower errors.

IV. Design Rationale

This section presents the design rationale of an FIB based on a learned index, referred to as a *learned FIB*, hereafter.

A. Goals and Approaches

Our goal is to realize fast and constant-time FIB lookup regardless of the length of matched prefixes.

It is of vital importance for fast FIB lookups that a lookup algorithm achieves sufficiently low time and space complexity. The designed algorithm, however, is sub-optimal in terms of complexity since the designed algorithm is faster than the optimal one on the assumed platform. Thus, we adopt, for example, the linear search algorithm rather than the exponential search algorithm for the local search phase of the learned index because it is faster for the designed learned FIB. In this way, our choice depends on a router platform, and therefore we describe the assumed router platform in the next section.

Our approach to realizing constant-time FIB lookup is to eliminate longest prefix matching. For instance, the computation time of trie-based FIBs [5], [6] generally increases with the length of matched prefixes because long prefixes correspond to vertices of high depth [4]. The computation time of hash-based FIBs [19] increases similarly because the matched prefix is iteratively searched for from the longest prefix to the shorter ones. In this way, longest prefix matching incurs the computation depending on the length of matched prefixes.

B. Assumed Router Platform

Recent advances in CPUs and fast networking technologies for computers make a software-based router, which is built on a computer without any TCAM devices, feasible, and we assume a computer with recent CPUs as a router platform.

The CPU supports advanced instruction sets, i.e., single instruction multiple data (SIMD) instructions (and advanced vector extensions (AVX) in the case of Intel CPUs [20]) and neural network instructions, such as the vectorized fused multiply-add (FMA) instruction [21]. SIMD instructions complete the same operation on multiple data simultaneously, thereby accelerating the computation speed. The CPU has a three-level cache system, which consists of a 1st-level instruction (L1I), a 1st-level data (L1D), a 2nd-level (L2), and a 3rd-level (L3) cache. Each CPU core has its exclusive L1I, L1D, and L2 cache, and all CPU cores share the L3 cache.

We need to optimize the algorithm and the data structure of a learned index to utilize the SIMD instructions. In addition, we need to eliminate pipeline stalls to optimize computation speed further. The major causes of pipeline stalls are categorized into the frontend bound, the backend bound, and the bad speculation [22]. Pipeline stalls in the three categories are mainly caused by instruction cache misses, data cache misses, and branch miss-predictions, respectively. We do not address pipeline stalls in the frontend bound since compilers often address the elimination of pipeline stalls in the frontend bound.

We eliminate any pipeline stalls in the bad speculation by implementing the algorithm of a learned index without any branches. We minimize the effects of pipeline stalls in the backend bound in the two steps. First, we place data on a memory device so that the hardware prefetcher of the CPU prefetches the data, thereby reducing data cache misses. Second, we try to minimize the latency due to pipeline stalls in case the data cannot be prefetched as the second step since the first step may not eliminate all the pipeline stalls in the backend bound. Specifically, we design a compact data structure to place as much data as possible on a higher-level cache.

C. Design Rationale behind Fast FIB Lookup

This section discusses the design rationale for fast lookup with a learned FIB.

1) Piece-wise Linear Approximation: In our view, the essential aspect of learned indexes is to approximate key-position relations in a KVS, i.e., the target function, rather than to learn the target function with machine learning. This finding implies that we can design an arbitrary approximation to the target function. It raises two questions: One is what kind

of approximation is suitable for the router platform, while the other is how to exploit the finding for realizing fast FIB lookup.

Regarding the first question, we choose a piece-wise linear approximation based on a combination of neural networks, considering the following two aspects. First, we adopt neural networks because we can optimize the implementation for the aforementioned router platform. In the next section, we discuss why we can optimize the neural network implementation for the assumed router platform. Second, we can inherit an important observation found by Rashelbach et al. [10] that a learned index consisting of neural networks with the ReLU activation function is a piece-wise linear function, which obviously means that a neural network with the ReLU activation function is also a piece-wise linear function. We develop a method to control the maximum prediction error of neural networks in a learned index based on the observation.

Rashelbach et al. [10] propose a learning approach using the piece-wise linear property of a neural network; however, the method cannot guarantee the maximum error. More precisely, they prove a theorem that the maximum prediction error of a learning index is analytically derived. Their proposal trains a neural network with a machine learning technique and analytically derives the maximum prediction error. If the maximum prediction error is larger than the pre-determined threshold, it increases the number of samples in the training data and retries the training. If it cannot achieve the maximum prediction error lower than the threshold, it increases the threshold and retries the training.

In contrast, our proposal first designs a piece-wise linear function so that the maximum prediction error is below the pre-determined threshold. Then, it determines the weights and the biases of neural networks so that the sequence of the piece-wise linear functions of the neural networks forms the designed piece-wise linear function. Thus, we derive the weights and the biases of neural networks in a deterministic manner rather than relying on learning algorithms, such as the backpropagation algorithm, and the maximum prediction error of the designed neural networks is bounded by the threshold.

The ability to control the maximum prediction error is an answer to the second question. First, it allows us to optimize the structure of a learned index, thereby reducing the computation time of a learned index. Second, it enables to implement the entire computation without any branches and loops, thereby eliminating pipeline stalls in the bad speculation. The two features are discussed in detail in Section IV-C3 and Section IV-C4, respectively.

2) Regression with Neural Networks: The prediction phase of learned indexes is a regression task. We choose neural networks for the regression task of learned FIBs rather than polynomial or nonlinear regression because we can optimize the implementation of a neural network on the aforementioned router platform. Specifically, we focus on the fact that the computation of a neural network can be programmed pwith SIMD instructions because it consists of element-wise vector multiplication, addition, and max operations. Furthermore, data necessary for computing a neural network, i.e., weights and biases, are vectors. This also contributes to vectorizing the neural network computation since a vector can be transferred to a SIMD register with a single instruction. In contrast, a polynomial regression task, for example, includes exponentiation computation, which is implemented with a sequence of instructions [23], and hence its computation speed may be worse than an optimized neural network.

Moreover, state-of-the-art CPUs have several features beneficial for neural networks, such as the vectorized fused multiply-add (FMA) instruction [21] and the half-precision floating point format [24], also referred to as bfloat16 (brain floating point). The FMA instruction computes the product of two numbers and adds another number to the product in a single instruction. The vectorized FMA instruction completes multiple FMA operations simultaneously. The instruction completes the computation of input values for multiple neurons simultaneously. The half-precision floating point format doubles the number of floating point numbers computed simultaneously compared to the case of single-precision floating point numbers. We choose the ReLU activation function rather than the sigmoid and the hyperbolic tangent activation function for the same reason. Specifically, it can be implemented with a SIMD instruction, i.e., the vectorized max operation.

Finally, in addition to the SIMD instructions, the cache system of CPUs supports the neural network computation. The weights and the biases of a neural network are accessed sequentially from the input layer to the output layer. We place the data continuously on a memory device, and hence the continuous placement allows the hardware prefetcher to prefetch them in advance so that pipeline stalls do not occur.

3) Simple Configuration of a Learned Index: We apply the neural network computation once, while the original recursive model index applies it multiple times depending on the stage depth, as illustrated in Fig. 1. One of the reasons why the original recursive model index increases the number of stages is to realize a better regression result. In contrast, we adopt a modified two-stage learned index, where the first-stage model is replaced with a table lookup operation while the second-stage model is designed neural networks, as shown in Fig. 2, because the maximum prediction error of a neural network is controlled. Specifically, the key space is equally divided into several sub spaces, and each model is responsible for a sub space. Each model approximates the target function in its responsible sub space. The m most significant bits of addresses, i.e., keys, are used to determine the responsible model. Thus, the proposal only incurs the computation of the bitwise shift operation and the one-time neural network computation.

In addition, we design a neural network with as few hidden layers and neurons as possible for realizing a given piece-wise linear approximation, thereby reducing the computation time and the size of the designed neural networks. The compactness of the designed neural networks contributes to placing the neural networks on a higher-level cache, thereby reducing the latency caused by pipeline stalls in the backend bound.

4) Branch- and Loop-free Implementation: Since the maximum prediction error is bounded, we can make the linear

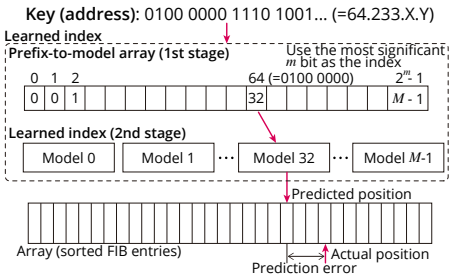


Fig. 2. Structure of a learned index for learned FIBs

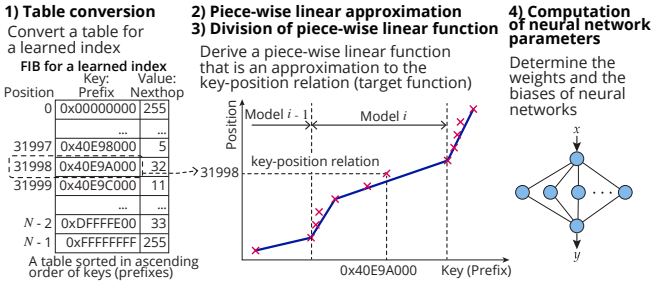


Fig. 3. Overview of the construction of a learned FIB

search algorithm faster than more sophisticated algorithms, such as the exponential search, of which time complexity is lower than the linear search algorithm. The linear search algorithm can be implemented without any branches by unrolled the loops in the search algorithm. In addition, the unrolled linear search can be programmed with SIMD instructions and therefore the computation time is also reduced. The implementation is discussed in Section VI.

D. Design Rationale behind Constant-time FIB Lookup

We avoid longest prefix matching as an approach to constant-time FIB lookup regardless of the length of matched prefixes. CIDR allows multiple prefixes with a common prefix into a single prefix, thereby reducing the size of FIBs [25]. On the one hand, the compactness of an FIB contributes to fast FIB lookup because it can be placed on a fast but small memory device, such as an SRAM device. On the other hand, the prefix aggregation incurs longest prefix matching. We stop using the prefix aggregation under the assumption of longest prefix matching. Instead, we convert a given FIB so that prefixes in FIB entries are disjoint to search for the prefixes with exact matching.

V. Learned FIB

A. Overview

This section presents the construction algorithm of a learned FIB. As illustrated in Fig.3, the construction of a learned FIB is composed of the following four steps:

- *Table conversion*: It converts an FIB so that entries in the FIB can be indexed with a learned index.
- *Piece-wise linear approximation*: It derives a piece-wise linear function that approximates the prefix-position relation with the maximum error lower than the predetermined threshold.
- *Division of the piece-wise linear function*: It divides the piece-wise linear function into sub-functions so that each sub-function is expressed with a specified neural network.
- *Computation of weights and biases of a neural network for each sub-function*: It determines the weights and the biases of a neural network to be identical to the sub-function.

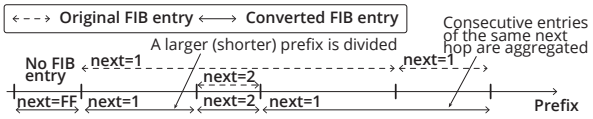


Fig. 4. An example of converting an FIB for indexing with a learned index

B. Table Conversion

The purpose of the table conversion is to eliminate longest prefix matching from the FIB lookup operation so that we can apply a learned index to an FIB. One way is to maintain the next hop information for all the addresses [9]. However, it is expensive because it has to compute the next hop for all 2^{32} addresses. We rethink this process. If a prefix contains any other prefixes, the prefixes are divided into disjoint prefixes, where no prefix contains any other prefixes. The list of resulting prefixes is similar to that of a leaf-pushed trie [8], where every prefix is located in leaves in the trie.

A schematic diagram of the table conversion is illustrated in Fig. 4. First, the boundaries of prefixes are sorted in ascending order. A prefix that contains another prefix is divided into several prefixes so that all the prefixes are disjoint. Dummy prefixes with dummy next hop information, e.g., a magic number 0xFF in Fig. 4, are inserted in the space where there are no corresponding prefixes. Finally, consecutive prefixes are aggregated if their next hop information is the same. The left boundary of each prefix is used as a representative key for the prefix. An FIB entry in a learned FIB is the pair of the left boundary of each prefix and its corresponding next hop information. For a given address, the next hop information is determined by searching for the prefix where the address belongs. Since the prefix where an address belongs is equivalent to the prefix nearest to and smaller than the address in the resulting FIB, the FIB lookup operation is realized with exact matching. The FIB entries are indexed by a learned index according to the procedures in the following subsections.

C. Piece-wise Linear Approximation

Let N and x_i be the number of prefixes and the ith prefix in the FIB constructed in the previous section. Because the prefixes are sorted in ascending order, the position of x_i is i. We refer to the function that traverses all the points $\mathcal{P} = \{(x_0, 0), (x_1, 1), \ldots, (x_{N-1}, N-1)\}$ in a two-dimensional

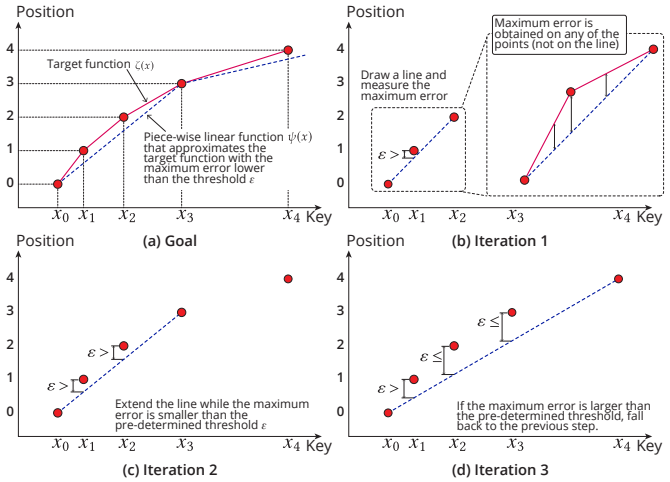


Fig. 5. Approximate the target function with a piece-wise linear function

space as the *target function*. We denote the target function and its piece-wise linear approximation by $\zeta(x)$ and $\psi(x)$, respectively. We describe a heuristic algorithm to derive a piece-wise linear approximation, $\psi(x)$, to the target function, $\zeta(x)$, with the maximum error lower than a pre-determined threshold, ε. That is, the algorithm derives $\psi(x)$ satisfying $|\zeta(x) - \psi(x)| \leq \varepsilon$ $(x_0 \leq x \leq x_{N-1})$.

The schematic of the algorithm is illustrated in Fig. 5, and the algorithm is summarized in Algorithm 1. The key idea behind the algorithm is to incrementally extend a line segment from a point (x_i, i) to another (x_j, j) $((x_i, i), (x_j, j) \in \mathcal{P}$, $i < j)$ with keeping the maximum error lower than the threshold, i.e., $|\zeta(x) - \psi(x)| \leq \varepsilon$ $(\forall x, x_i \leq x \leq x_j)$, as illustrated in Fig.5(b)–(d). If the maximum error is greater than ε, the algorithm terminates the line segment at $(x_{j-1}, j-1)$ and draws a new line segment from $(x_{j-1}, j-1)$ to another. The error $|\zeta(x) - \psi(x)|$ takes the maximum value on one of the points in $\mathcal{P}$, as shown in Fig. 5(b). We therefore only examine the error on the points in $\mathcal{P}$.

Algorithm 1: Deriving a piece-wise linear function

Input: $\mathcal{P}$: Points on the target function, ε: The threshold of the maximum error
Output: $\mathcal{B}$: Boundaries of line segments of the piece-wise linear approximation

$\mathcal{B} \leftarrow \{(x_0, 0)\}$
// l, r: left and right boundary of a line segment
$l \leftarrow 0$, $r \leftarrow 2$
while $r < N$ **do**
  // Derive a line passing through (x_l, l) and (x_r, r)
  $a \leftarrow (r-l)/(x_r - x_l)$, $b \leftarrow l - a \times x_l$
  // Examine the error between x_{l+1} and x_{r-1}
  for $i \leftarrow l+1$ **to** $r-1$ **do**
    $p \leftarrow a \times x_i + b$ // Compute the y-value on the line for the x-value of x_i
    if $|p - i| > \varepsilon$ **then**
      // The error is larger than the threshold
      Append $(x_{r-1}, r-1)$ to $\mathcal{B}$ // Append the previous point to the set of boundaries
      $l \leftarrow r-1$
      break
  $r \leftarrow r+1$

D. Division of a Piece-wise Linear Function

A piece-wise linear function of n line segments can be expressed as a neural network of a hidden layer of n neurons, as we will explain in the next section. We therefore need to divide the piece-wise linear function $\psi(x)$ obtained in the previous step into sub-functions, each of which consists of at most n line segments.

The proposed learned index uses the m most significant bits of prefixes and addresses to select models, as explained in Section IV-C and Fig. 2. Consequently, the ith model is responsible for the range $[2^{32-m}i, 2^{32-m}(i+1) - 1]$. Thus, the piece-wise linear function must be divided along the boundaries of the ranges. We derive the value of m so that there are at most n line segments of the given piece-wise linear function for all the ranges $[2^{32-m}i, 2^{32-m}(i+1)-1]$ $(\forall i, 0 \leq i \leq 2^m - 1)$.

This division policy should coincide with the efficient implementation of the lookup program such that the number of neurons of all the neural networks should be identical. This eliminates branches in the program, thereby contributing to the fast computation. We use a neural network of one hidden layer of n neurons for all the models. However, in the naive division, while there are n line segments in some ranges, there are fewer line segments in the other ranges. It is obviously inefficient to assign n neurons to express a piece-wise linear function having less than n line segments. We therefore merge sub-functions so that the merged sub-function has as many line segments as possible with satisfying the condition of less than or equal to n line segments.

E. Computation of Neural Network Parameters

In this section, we first explain that a neural network is a piece-wise linear function, and then we describe how to derive the weights and the biases so that the neural network and a given piece-wise linear function are identical.

1) Piece-wise Linear Function of Neural Network: A neural network of one hidden layer of n neurons is formulated as

$$y = \sum_{k=1}^{n} \mathrm{ReLU}\left(w_{1k}x + b_{1k}\right) w_{2k} + b_2, \tag{1}$$

where x and y are the input and the output, w_{1k} and b_{1k} are the weight and the bias for the hidden layer, and w_{2k} and b_2 are the weight and the bias for the output. The ReLU activation function, $\mathrm{ReLU}(z)$, is equivalent to $\max(z, 0)$.

The neural network is the summation of the n functions, $f_k(x)$ $(k = 1, \ldots, n)$, where $f_k(x)$ denotes $f_k(x) =$

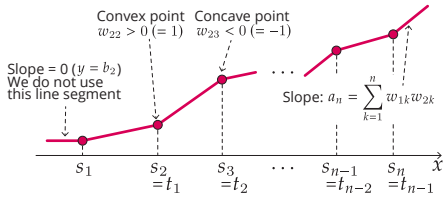


Fig. 6. An example of a piece-wise linear function of a neural network

ReLU $(w_{1k}x + b_{1k})\, w_{2k} = w_{2k} \max(w_{1k}x + b_{1k}, 0)$. The function $f_k(x)$ consists of two linear functions:

$$f_k(x) = \begin{cases} w_{2k}(w_{1k}x + b_{1k}) & \text{if } w_{1k}x + b_{1k} \geq 0 \\ 0 & \text{otherwise} \end{cases}. \quad (2)$$

Let s_k denote the solution of $w_{1k}x + b_{1k} = 0$, i.e., $s_k = -b_{1k}/w_{1k}$. Because the two functions in (2) change only at $x = s_k$, the neural network is a piece-wise linear function of $n + 1$ lines segmented at $\{s_1, \ldots, s_n\}$. An example of a piece-wise linear function of a neural network is illustrated in Fig. 6. Without loss of generality, we assume that the functions $f_k(x)$ are sorted in ascending order of s_k and all the solutions are different, i.e., $s_1 < s_2 < \cdots < s_n$, because we need to express a piece-wise linear function with a neural network of as few neurons as possible. Note that the slope of the piece-wise linear function is zero in the range of $f_k(x) = 0$ for all k. We therefore use the remaining n line segments to express a given piece-wise linear function with the neural network.

2) Computing Parameters of Neural Network: Next, we describe how to determine the weights and the biases of the neural network so that the piece-wise linear function of the neural network and a given piece-wise linear function are identical. To avoid confusion, we refer to a piece-wise linear function of a neural network as a neural network function.

The given piece-wise linear function is defined as follows: It is composed of n line segments, and the boundaries of the line segments are $\{(t_1, u_1), (t_2, u_2), \ldots, (t_{n-1}, u_{n-1})\}$. The slope of the kth line segment is a_k. The target function is monotonically and strictly increasing, and hence its piece-wise linear approximation is also monotonically and strictly increasing, i.e., $a_k > 0$ $(\forall k)$.

We next discuss how to determine a neural network function that is identical to the piece-wise linear function. The weights for the hidden layer w_{1k} must be positive so that the neural network function is a monotonically and strictly increasing function. Specifically, there are four cases of changing the slope of $f_k(x)$ depending on the sign of w_{1k} and w_{2k}, as shown in Fig. 7. The line segment of slope 0 must be located in the leftmost or the rightmost side of the neural network function to ignore the line segment. If the sign of the weights for the hidden layer w_{1k} is positive, the slope of the left-side line segment of $f_k(x)$ is zero. Conversely, if the sign is negative, the slope of the right-side line segment is zero. Hence, the signs of w_{1k} $(\forall k)$ must be identical. Otherwise, the slope of one of the intermediate line segments might be zero, which does not satisfy the condition of strictly increasing.

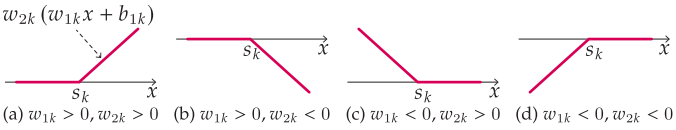


Fig. 7. The four patterns of a ReLU function

For simplicity, we use positive values for w_{1k}. In this case, (2) is expressed without the ReLU function as

$$y = \begin{cases} b_2 & \text{if } x < s_1 \\ w_{21}\,(w_{11}x + b_{11}) + b_2 & \text{if } s_1 \leq x < s_2 \\ \cdots & \\ \sum_{k=1}^{i} w_{2k}\,(w_{1k}x + b_{1k}) + b_2 & \text{if } s_i \leq x < s_{i+1} \\ \cdots & \\ \sum_{k=1}^{n} w_{2k}\,(w_{1k}x + b_{1k}) + b_2 & \text{if } s_n \leq x \end{cases}. \quad (3)$$

This equation indicates that the slope of the ith line segment is $\sum_{k=1}^{i} w_{1k}w_{2k}$. Note that we ignore line segments of slope 0, and the first line segment is $w_{21}\,(w_{11}x + b_{11}) + b_2$ $(s_1 \leq x < s_2)$. Hence, the intersection between the $(i-1)$th and the ith line segment is a convex (concave) point if w_{2i} is positive (negative). Thus, we can express an arbitrary piece-wise linear function that is strictly increasing with a neural network.

According to the aforementioned discussion, the weights and the biases of the neural network function are determined as follows: First, w_{11} and w_{21} are set to $w_{11} = a_1$ and $w_{21} = 1$, respectively. Next, for the slope of the ith line segment $(i > 1)$, we set w_{2i} to either $+1$ or -1 depending on whether the intersection between the $(i-1)$th and the ith line segment is convex or concave and we set w_{1k} to $w_{1k} = |a_{k-1} - a_k|$. Then, we obtain the bias for the hidden layer b_{1k} as $b_{1k} = -t_k w_{1k}$, which is the solution of $s_k = t_k$. Finally, the bias for the output layer b_2 is determined so that the neural network function perfectly fits the piece-wise linear function. The simplest way is to solve the equation $w_{21}\,(w_{12}t_1 + b_{12}) + b_2 = u_1$ because $s_2 = t_1$, as illustrated in Fig. 6.

VI. Implementation of Learned FIB

This section presents the implementation of the lookup algorithm for a learned FIB.

The algorithm is summarized in Algorithm 2. The algorithm consists of the prediction phase (line 1–7) and the local search phase (line 8–11). The algorithm is fed with an address as an input and returns the next hop information as the output. The data structures of a learned FIB are illustrated in Fig. 8.

We implement the algorithm according to the design rationale in Section IV. The entire program is implemented without any loops and branches. First, the algorithm extracts the m most significant bits of a given address with the shift right logical instruction (line 1). Next, it obtains the model corresponding to the address using the extracted prefix (line 2–3). The neural network computation (line 4–7) is implemented with the SIMD (vectorized) versions of the FMA (vfmadd), the max (vmax), and the multiplication (vmul) instruction. The accumulation operation, which computes the

Algorithm 2: Algorithm for lookup the learned FIB

Input: x: Address (32-bit unsigned integer)
Output: h: Next hop information
// **— Prediction phase —**
1 $r \leftarrow \mathrm{srl}(x, 32 - m)$ // Shift (slr: shift right logical)
2 $i \leftarrow \mathrm{table}[r]$ // Get model index
3 $\boldsymbol{w}_1, \boldsymbol{b}_1, \boldsymbol{w}_2, b_2 \leftarrow \mathrm{model}[i]$ // Get weights and biases
// Predict the position of x via a neural network
4 $\boldsymbol{z} \leftarrow \mathrm{vfmadd}(\boldsymbol{w}_1, x, \boldsymbol{b}_1)$ // $\boldsymbol{z} \leftarrow \boldsymbol{w}_1 \times x + \boldsymbol{b}_1$
5 $\boldsymbol{z} \leftarrow \mathrm{vmax}(\boldsymbol{z}, 0)$ // $\boldsymbol{z} \leftarrow \mathrm{ReLU}(\boldsymbol{z})$
6 $\boldsymbol{z} \leftarrow \mathrm{vmul}(\boldsymbol{w}_2, \boldsymbol{z})$ // Element-wise $\boldsymbol{w}_2 \times \boldsymbol{z}$
7 $y \leftarrow \mathrm{accumulate}(\boldsymbol{z}) + b_2$ // $y \leftarrow \sum \boldsymbol{z} + b_2$
// **— Local search phase —**
8 $l \leftarrow y - \varepsilon$ // Search the range $[y - \varepsilon, y + \varepsilon]$ for x
// Set the results of $\boldsymbol{k}[l : l + 2\varepsilon] \le x$ to $\boldsymbol{z}$
9 $\boldsymbol{z} \leftarrow \mathrm{vpcmp}(\boldsymbol{k}[l : l + 2\varepsilon], x)$
10 $l \leftarrow l + \mathrm{accumulate}(\boldsymbol{z})$
11 $h \leftarrow \mathrm{nexthop}[l]$

sum of elements in a vector, is also implemented with several SIMD instructions without loops.

The local search phase is implemented without any loops and branches (line 8–11). As explained in Section V-B, the local search is completed when it finds the prefix nearest to and smaller than a given address. Additionally, the position of the prefix is in the range of $[y - \varepsilon, y + \varepsilon]$ because the maximum prediction error is ε. Hence, we can derive the position of the prefix by counting the number of prefixes less than or equal to the address in the range and adding the number to $y - \varepsilon$. The counting operation can be vectorized with the vpcmp instruction family, which compares multiple values simultaneously and stores the comparison results (0 and 1) into a vector. The algorithm obtains the position of the prefix by accumulating the comparison results.

Figure 8 depicts the data structures of learned FIB. The data structures are also optimized by taking into account the following two points. One is that the weights and the biases of a neural network are continuously placed on a memory device to be prefetched by the hardware prefetcher. The other is that the array of keys (prefixes) and that of values (next hop information) are decoupled to implement the local search phase with SIMD instructions. An alternative design is implementing the FIB as an array of compositions (structs in C/C++) of a key and a value. Our design is superior to the alternative design because keys are continuously placed on a memory device. The keys in the range of $[y - \varepsilon, y + \varepsilon]$ can be transferred to a SIMD register in a single instruction. The design has another benefit: the number of memory accesses is reduced, thereby reducing the possibility of evicting necessary data from the L2 cache. While 2ε elements in the array of keys are accessed during the local search phase, a single element alone is accessed in the array of next hop information.

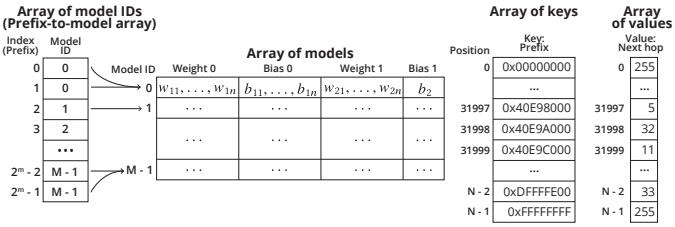


Fig. 8. The data structures of a learned FIB

VII. Open Issues

There are several open issues for the learned FIB. A crucial issue is its expensive FIB update cost due to the following three reasons. First, the table conversion is similar to the leaf-pushing technique, and hence, it spends the computation time depending on the number of prefixes. In particular, the insertion and the deletion of a short prefix are likely to take long time because a short prefix tends to cover many prefixes. It may be a promising approach to circumventing the expensive FIB update to introduce sophisticated leaf-pushing techniques [26]. Second, insertion and deletion of an FIB entry also require time depending on the number of prefixes because FIB entries are stored in a sorted array. Finally, it requires updating neural networks if the maximum error of the previously designed neural networks exceeds the predetermined threshold ε.

Another open issue is to apply the proposed design for learned indexes to other tables, such as IPv6 FIBs. Though the core ideas of the proposal, such as the piece-wise linear approximation and the branch-free implementation, are applicable to tables in other domains, the current design is highly optimized for IPv4 FIBs. For instance, the implemented learned FIB fits into the assumed router platform well since the size of keys, i.e., prefixes, is 32 bits. Inventive ideas are required to apply the proposed design to 128-bit IPv6 addresses on top of the assumed router platform.

VIII. Performance Evaluation

A. Overview of Performance Evaluation

The objectives of the performance evaluation are threefold. First, we validate that the learned FIB realizes fast and constant-time FIB lookup regardless of the length of matched prefixes by comparing it with a state-of-the-art FIB based on Poptrie [6] (Poptrie FIB) in Section VIII-C. Second, we validate that the following two learned index designs are beneficial for fast FIB lookups via comparisons with two variants of FIBs based on a learned index in Section VIII-D. One is to validate the use of the table lookup operation in the first stage. We use an FIB with the original RMI (RMI FIB) as a comparison. The RMI FIB uses neural networks in all the stages, whereas our learned index uses neural networks only in the last stage. Note that we use a two-stage RMI since our learned index has two stages. The other is to validate the use of the piece-wise linear approximation for constructing neural networks. We use the FIB proposed in our previous

study [9], which is similar to the proposal except that it construct neural networks using a machine learning technique. We refer to this FIB as the machine learning (ML) FIB. Finally, we analyze performance characteristics of the learned FIB in Section VIII-E.

B. Evaluation Conditions

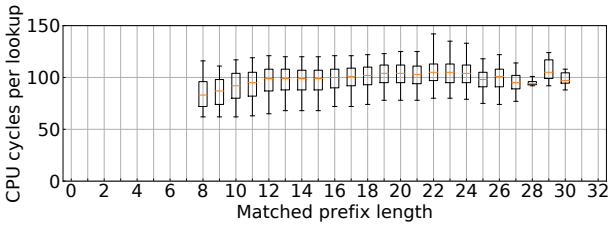
While we implement the learned, the ML, and the RMI FIB in C++, we use the open source software of the Poptrie FIB implemented in C [27]. The maximum error ε and the number of neurons n in a hidden layer are 32 and 8, respectively. We evaluate effects of these parameters in Section VIII-E.

We use a server with an Intel Xeon Gold 6130 CPU and 128 Gbytes DDR4 DRAMs as an experiment platform. All the programs for the FIBs run on a single thread. The CPU has a 32 Kbytes L1D cache and a 1.25 Mbytes L2 cache in each CPU core, and a 22 Mbytes L3 cache shared by all CPU cores. The L1D, the L2, and the L3 cache require 5, 12, and 38 cycles to be accessed, respectively [28].

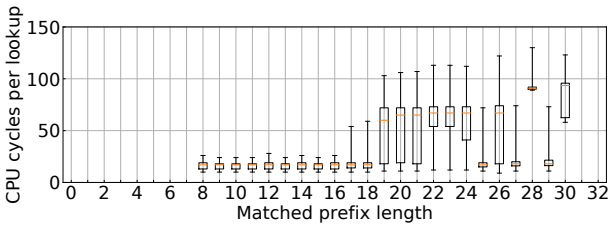
We use a real traffic trace as well as a random traffic pattern. The real traffic trace was captured by CAIDA at an Equinix datacenter in New York [29]. Under the random traffic pattern, randomly generated addresses are looked up. We mainly use the random traffic pattern since various properties, such as the spatial and the temporal address locality, are often observed in real traffic traces. Such locality affects the lookup performance. We use the real traffic trace for the evaluation in Fig. 10 in Section VIII-C, which aims at analyzing the CPU cache locality.

FIBs are created from a BGP routing information base (RIB) snapshot provided by the Route Views project [30]. We use the snapshot captured on Nov. 20th, 2019. The RIB has 360879 prefixes, and prefixes whose length is longer than or equal to 24 account for 33%. The learned, the Poptrie, and the ML FIB are 1.83 Mbytes, 1.98 Mbytes, and 1.78 Mbytes, respectively.

We use the computation time spent for a single lookup operation as a performance metric. The computation time is measured as the number of CPU cycles using the read time stamp counter (RDTSC) instruction.

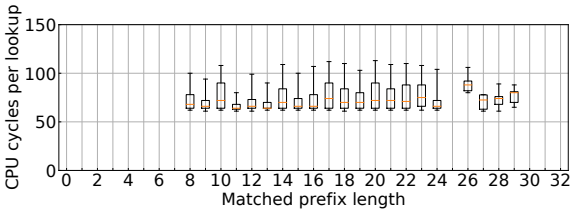


(a) Learned FIB

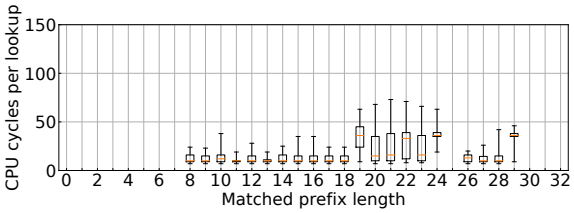


(b) Poptrie FIB

Fig. 9. The computation time of the learned FIB and the Poptrie FIB for each prefix length in the case of the random traffic pattern



(a) Learned FIB



(b) Poptrie FIB

Fig. 10. The computation time of the learned FIB and the Poptrie FIB for each prefix length in the case of the real traffic trace

C. Comparison Against Longest Prefix Matching

First, we evaluate the learned FIB realizes fast and constant-time FIB lookup by comparing it with the Poptrie FIB. The computation time for each length of matched prefixes under the random traffic pattern is plotted in Fig. 9. The wick of each candlestick, the body, and the internal bar represent the 5th/95th percentile, the first and the third quartile, and the median, respectively.

This figure provides the following two observations: First, the learned FIB realizes constant-time lookup regardless of the length of matched prefixes, whereas the computation time of the Poptrie FIB increases if the prefix length is longer than 18. The learned FIB always performs the prediction phase, the computation of a neural network, and the local search phase, the linear search for a constant range (2ε). In contrast, the Poptrie FIB requires traversing vertices of a trie if the matched prefix is long, and the traversal increases with the matched prefix length.

Second, the Poptrie FIB outperforms the learned FIB in terms of the average computation time, especially in the case that the length of matched prefixes is short, because the Poptrie FIB completes a lookup operation by a single access to an array. Specifically, the array maintains 2^{18} entries indexed by the significant 18 bits of prefixes, and the entries have next hop values. Hence, the lookup operation does not need to traverse any vertices of the Poptrie if the length of matched prefixes is less than 18.

These results suggest that using the learned FIB is beneficial in the case that long prefixes, such as IPv6 prefixes and 24-bit IPv4 prefixes, dominate in an FIB. As a future insight, the number of long prefixes will grow up due to the need for fine-

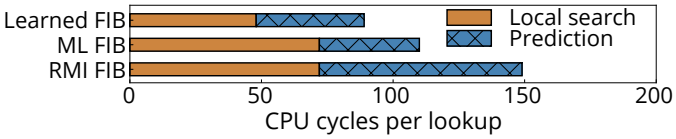


Fig. 11. The computation time of the learned, the ML, and the RMI FIB.

grained traffic control. The trend implies that the learned FIB will be useful in the near future.

We next evaluate the lookup speed with the real traffic trace in Fig. 10. The computation time of both the learned and the Poptrie FIB decreases compared to the case of the random traffic pattern due to the high CPU cache locality. Since there are many consecutive accesses to the same destination addresses, data for the FIBs tend to be accessed from a higher level CPU cache.

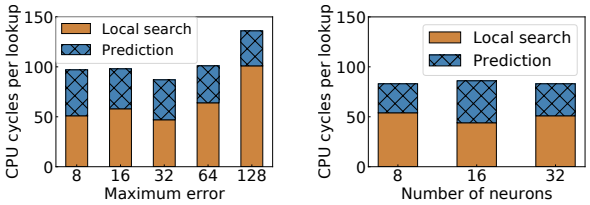
D. Comparison Against Machine Learning

We compare the computation time of the learned FIB with the ML and the RMI FIB to prove that our designed learned index improves the computation time. Figure 11 shows the average computation time. The learned FIB reduces the computation time by 19.1% and 40.0% compared to the ML and the RMI FIB, respectively.

The three FIBs similarly consist of the prediction and the local search phase. To prove the benefits of our design in greater detail, we further decompose the computation time of a single lookup operation into the prediction and the local search phase. The RMI FIB spends more time for the prediction phase than the learned and the ML FIB. The RMI FIB needs to compute neural networks twice in the prediction phase, while the learned and the ML FIB need the computation once. This result indicates that replacing a neural network with a table lookup operation in the 1st stage of the learned index improves the computation speed.

The difference in the local search phase between the learned and ML FIB comes from our design choice that the piece-wise linear approximation bounds the maximum error ε. This allows the learned FIB to complete the local search phase without any branches and loops. In contrast, the ML FIB must perform the exponential search algorithm, which contains loops and branches, to correct the prediction error since ε cannot be determined in advance.

We quantitatively investigate the impact of the difference on the computation time with the Intel Vtune profiler [31]. We measure how much the computation time wasted by pipeline stalls accounts for the entire computation time of the local search phase. In the case of the ML FIB, pipeline stalls due to branch prediction misses consume 44.4% of the entire computation time, whereas there is no pipeline stalls due to branch prediction misses in the case of the learned FIB. The analysis proves that the learned FIB successfully eliminates branches and loops and the elimination contributes to the improvement in the computation time.



(a) Lookup time vs. maximum error

(b) Lookup time vs. neurons

Fig. 12. The computation time when varying the maximum error and the number of neurons.

E. Impacts of Parameters for Learned FIB

Finally, we analyze effects of the parameters on the computation time of the learned FIB. Two key parameters to be analyzed in the learned FIB are the maximum error ε and the number of neurons n in each neural network.

Figure 12a shows the average computation time of a single lookup operation when varying ε from 8 to 128. Though the computation time of the local search phase decreases with ε, that of the prediction phase increases. While the range of the array of prefixes to be searched decreases with ε, the number of models to be stored in the array of models increases. Hence, models do not fit into the L1D cache if ε is small. This result suggests that $\varepsilon = 32$ is a good trade-off point from the perspective of the performance of these two phases.

We next evaluate the computation time when varying the number of neurons n from 8 to 32. The result is plotted in Fig. 12b. In this evaluation, we set ε to 32. Although the computation time of the prediction phase slightly increases in the case of $n = 32$, the increase is negligible. This is because that the CPU supports the simultaneous computation of up to 32 neurons. The CPU, which supports the AVX-512 extension [20], has SIMD registers of 512 bits, hence supporting the simultaneous computation of up to 16 single-precision (32-bit) floating points. In addition, each CPU core of the CPU has two execution units used for the neural network computation [28]. The CPU therefore supports the computation of a neural network of a hidden layer of 32 neurons.

IX. Conclusion

In this paper, we have proposed an FIB based on a learned index, aiming at a compact FIB representation and fast FIB lookups. We have designed a near-optimal implementation of a learned index for a recent computer platform, leveraging that it is sufficient to approximate the key-position pairs in the FIB with a piece-wise linear function. We have demonstrated that the learned FIB realizes fast and constant-time FIB lookups using real BGP routing information snapshots. Finally, we have applied a learned index to an FIB at the cost of efficient FIB updates, which is deferred for our future work.

Acknowledgement

The authors are grateful to our shepherd, Gábor Rétvári, and the anonymous reviewers for their constructive comments and suggestions.

References

[1] A. Asthana, C. Delph, H. V. Jagadish, and P. Krzyzanowski, "Towards a gigabit IP router," *Journal of High Speed Networks*, vol. 1, no. 4, pp. 281–288, Oct. 1992.

[2] C. Partridge, P. P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. C. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. D. Troxel, D. Waitzman, and S. Winterble, "A 50-Gb/s IP router," *IEEE/ACM Transactions on Networking*, vol. 6, no. 3, pp. 237–248, Jun. 1998.

[3] D. Shah and P. Gupta, "Fast updating algorithms for TCAM," *IEEE Micro*, vol. 21, pp. 36–47, Jan./Feb. 2001.

[4] H. J. Chao and B. Liu, *High performance switches and routers*. John Wiley & Sons, Inc., 2007.

[5] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083–1092, Jun. 1999.

[6] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup," in *Proceedings of ACM SIGCOMM*, Aug. 2015, pp. 57–70.

[7] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of ACM SIGMOD*, Jun. 2018, pp. 489–504.

[8] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Transactions on Computer Systems*, vol. 17, no. 1, pp. 1–40, Feb. 1999. [Online]. Available: https://doi.org/10.1145/296502.296503

[9] S. Higuchi, J. Takemasa, Y. Koizumi, A. Tagami, and T. Hasegawa, "Feasibility of longest prefix matching using learned index structures," *ACM SIGMETRICS Performance Evaluation Review*, vol. 48, no. 4, pp. 45–48, May 2021. [Online]. Available: https://doi.org/10.1145/3466826.3466842

[10] A. Rashelbach, O. Rottenstreich, and M. Silberstein, "A computational approach to packet classification," in *Proceedings of ACM SIGCOMM*, Aug. 2020, pp. 542–556.

[11] A. J. McAuley and P. Francis, "Fast routing table lookup using CAMs," in *Proceedings of IEEE INFOCOM*, Mar. 1993, pp. 1382–1391.

[12] H. Fadishei, M. S. Zamani, and M. Sabaei, "A novel reconfigurable hardware architecture for ip address lookup," in *Proceedings of ACM/IEEE ANCS*, Oct. 2005, pp. 81–90.

[13] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee IP lookup performance with FIB explosion," in *Proceedings of ACM SIGCOMM*, Aug. 2014, pp. 39–50.

[14] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," in *Proceedings of ACM SIGCOMM*, Sep. 1997, pp. 3–14.

[15] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proceedings of IEEE INFOCOM*, Mar. 1998, pp. 1240–1247.

[16] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: Hardware/software ip lookups with incremental updates," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 97–122, Apr. 2004.

[17] M. Zec, L. Rizzo, and M. Mikuc, "DXR: towards a billion routing lookups per second in software," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, pp. 29–36, Sep. 2012.

[18] RIPE Labs, "Visibility of IPv4 and IPv6 prefix lengths in 2019." [Online]. Available: https://labs.ripe.net/author/stephen_strowes/visibility-of-ipv4-and-ipv6-prefix-lengths-in-2019/

[19] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Transactions on Networking*, vol. 14, pp. 397–409, Apr. 2006.

[20] Intel Corporation, "Intel advanced vector extensions 512 (Intel AVX-512)." [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html

[21] ——, "Intel 64 and IA-32 architectures software developer's manual," Apr. 2021.

[22] A. Yasin, "A top-down method for performance analysis and counters architecture," in *Proceedings of IEEE ISPASS*, Mar. 2014, pp. 35–44.

[23] Intel Corporation, "Intel intrinsics guide." [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[24] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A study of BFLOAT16 for deep learning training," *arXiv preprint: arXiv:1905.12322v3*, Jun. 2019.

[25] B. Zhang, L. Wang, X. Zhao, Y. Liu, and L. Zhang, "FIB aggregation," Internet Engineering Task Force, Internet-Draft draft-zhang-fibaggregation-02, Oct. 2009, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-zhang-fibaggregation-02

[26] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: Towards entropy bounds and beyond," in *Proceedings of ACM SIGCOMM*, 2013, pp. 111–122. [Online]. Available: https://doi.org/10.1145/2486001.2486009

[27] H. Asai, "An implementation of poptrie IP routing table lookup algorithm." [Online]. Available: https://github.com/pixos/poptrie

[28] Intel Corporation, "Intel 64 and IA-32 architectures optimization reference manual," May 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html

[29] CAIDA, "The CAIDA anonymized internet traces dataset." [Online]. Available: https://www.caida.org/catalog/datasets/passive_dataset/

[30] University of Oregon Route Views Project, http://www.routeviews.org/routeviews/.

[31] Intel Corporation, "Intel VTune profiler." [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html