IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 6, 2020, accepted August 31, 2020, date of publication September 10, 2020, date of current version September 23, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3023156

Fast Longest Prefix Matching by Exploiting SIMD Instructions

YUKITO UENO[1,3], RYO NAKAMURA[2], YOHEI KUGA[2], AND HIROSHI ESAKI[1], (Member, IEEE)
[1]Graduate School of Information Science and Technology, The University of Tokyo, Tokyo 113-0033, Japan
[2]Information Technology Center, The University of Tokyo, Tokyo 113-0033, Japan
[3]Innovation Center, NTT Communications, Tokyo 108-8118, Japan

Corresponding author: Yukito Ueno (eden@g.ecc.u-tokyo.ac.jp)

ABSTRACT Longest prefix matching (LPM) is a fundamental process in IP routing used not only in traditional hardware routers but also in software middleboxes. However, the performance of LPM in software is still insufficient for processing packets at over 100 Gbps, although previous studies have tackled this issue by exploiting the CPU cache or accelerators such as GPUs. To improve the performance of software LPM further, we propose a novel LPM method called Spider, which exploits a single-instruction multiple-data (SIMD) mechanism in the CPU. Spider achieves performing LPM for up to 16 destination IP address in parallel by a routing table structure carefully designed for processing by the SIMD instructions. We evaluated Spider from the following three perspectives: the improvement of LPM performance derived from the parallelism provided by the SIMD mechanism, performance comparison with other methods, and performance scalability. The evaluation shows that Spider dramatically improves the LPM performance, which reaches 1.8–3.2 times compared with the state-of-the-art methods. Moreover, Spider achieves 5,074 million lookups per second with 16 CPU cores, which is equivalent to the processing capacity of 3.4 Tbps in short packets; the performance opens up the possibility of packet processing at the terabit-class rate by software.

INDEX TERMS IP routing, longest prefix matching (LPM), single-instruction multiple-data (SIMD), software middlebox.

I. INTRODUCTION

Longest prefix matching (LPM) is a fundamental process of IP routing in both hardware routers and software middleboxes. In recent commercial IP networks, 100 Gbps is a mainstream interface speed, and 400 Gbps has been emerging to deliver terabit-class service traffic. Thus, both hardware routers and software middleboxes need to perform LPM at a speed that can accommodate multiple of the 100 Gbps interfaces. Hardware routers have already been able to perform LPM at that speed using dedicated mechanisms such as ternary content addressable memory (TCAM) or a network processor unit. On the other hand, software LPM cannot deliver as much performance as hardware, although software middleboxes are actively used for various use cases, e.g., network function virtualization (NFV) [1], [2], software routers for backbone networks [3], and software-defined WAN [4]. Because the software middleboxes need to process LPM in software, improving the performance of LPM in software is the key to achieve these use cases while satisfying the communication speed requirement.

A major approach for fast LPM in software is to shorten the time for looking up a destination IP address by leveraging the CPU cache to minimize the latency for accessing data [5]–[12]. Although their performance has not reached the speed of a multiple of the 100 Gbps interfaces yet, their further performance improvement would be limited because they have thoroughly exploited the CPU cache, and so their remaining improvement factor is the increase of the CPU frequency, which has stagnated [13].

To clarify the potential of the state-of-the-art methods for faster LPM, we measured and estimated the LPM performance of the methods based on CPU frequencies. Figure 1 shows the performance of the recent LPM methods, i.e., Poptrie [10], DXR [11], and our LPM method on a single CPU core, along with the CPU frequency changes. The x-axis and y-axis indicate the LPM performance in a million lookups per second (Mlps) and latency for a single LPM iteration, respectively, according to the CPU frequencies. The points at 1.0, 2.1, and 3.7 GHz are measured values in

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Masini.

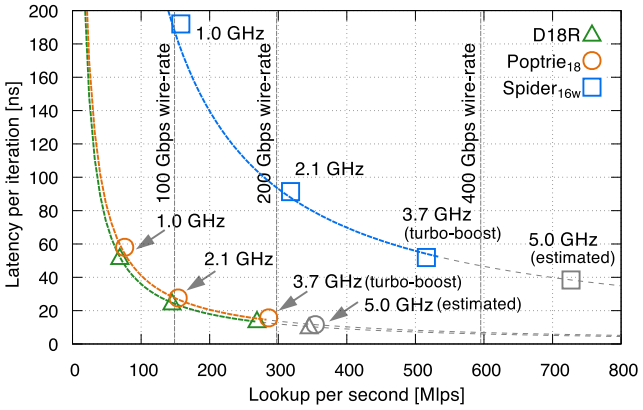

FIGURE 1. The performance of the recent LPM methods and our LPM method along with the CPU frequency changes. The lookup rate of the proposed LPM method named Spider is estimated to overcome 400 Gbps wire-rate when the CPU frequency reaches 5.0 GHz. The numbers attached to each method name in the legend mean method-specific settings, which are described in Section V.

Section V-D and V-E. Moreover, based on the CPU cycles per LPM iteration of each method (measured in § V-D) and the constant increase of the lookup rate of each method to CPU frequencies (measured in § V-E), we plot estimated lines and values if the CPU frequency is 5.0 GHz. The formulas for the estimation are

$$\begin{aligned}&\text{Latency per iteration (ns)}\\&\quad=\frac{1\ \text{(GHz)}}{t\ \text{(GHz)}}\times\text{CPU cycles per iteration},\\&\text{Lookup rate (Mlps)}\\&\quad=t\ \text{(GHz)}\times\frac{\text{Lookup rate at 1 GHz (Mlps)}}{1\ \text{(GHz)}},\end{aligned}$$

where t (GHz) is the arbitrary CPU frequency. Besides, because the latency per iteration and the lookup rate are mediated by the t (GHz), the estimated performance along with the CPU frequency changes can be expressed by

$$\begin{aligned}&\text{Latency per iteration (ns)}\\&=\frac{\text{CPU cycles per iteration}\times\text{Lookup rate at 1 GHz (Mlps)}}{\text{Lookup rate (Mlps)}}.\end{aligned}$$

The equation indicates that the latency per iteration is inversely proportional to the lookup rate, as shown in Figure 1. Therefore, when the latency per iteration of the LPM methods is lower, the lookup rate would be higher.

The figure indicates that the state-of-the-art methods minimizing the latency per iteration face an asymptotic limit on the latency. The thorough optimization focusing on the use of the CPU cache results in a situation where the processing of the CPU instructions is the bottleneck, which never disappears. Moreover, the measured and estimated values in the figure suggest that the increase of the CPU frequency does not bring sufficient performance gain for the methods. The lookup rates at 3.7 GHz of the Poptrie$_{18}$ and D18R do not overcome the 200 Gbps wire-rate. If the CPU frequency becomes 5.0 GHz, their performance gain is still insufficient to achieve the 400 Gbps wire-rate, although further performance improvement on their approach relies on the increase of the CPU frequency. Using multiple CPU cores certainly increases the total lookup rate of LPM by a dozen times, according to the number of CPU cores. This approach is practical; however, it requires distributing multiple flows among multiple cores. Thus, the throughput of a single flow cannot achieve 200 and 400 Gbps link speeds if the lookup rate of LPM on a single CPU core is insufficient.

To overcome the limit of the recent methods, we propose a novel LPM method called Spider, which exploits the single-instruction multiple-data (SIMD) mechanism in the CPU. The approach of Spider is to parallelize the LPM procedure inside a single CPU core to increase the throughput for lookup, not to minimize the latency for it, as in previous work. A single iteration for LPM in Spider processes up to 16 destination IP addresses at the same time. As shown in Figure 1, Spider achieves far higher lookup rates than other methods while requiring longer latency to process a single lookup as a CPU-based method due to the characteristics of SIMD instructions. When the CPU frequency reaches 5.0 GHz, Spider is estimated to overcome the 400 Gbps wire-rate.

This work introduces new contributions in three main points, although our previous work [14] has already proposed the basic concept of Spider. First, we provide a more detailed explanation of the concept of Spider (§ III). Second, we discuss considerations when applying Spider to real-world packet processing applications (§ IV). In that section, we show that Spider can support IPv6 LPM while keeping its performance advantage. Third, we demonstrate the usefulness of our approach by intensive evaluations of Spider and recent software LPM methods from multiple perspectives (§ V). The evaluation is also extended from our previous work to reveal more detailed characteristics of Spider, including the applicability to real-world packet processing applications (§ V-C), the CPU cycles to process the lookup procedure (§ V-D), and the performance under different CPU frequencies (§ V-E). In summary, the new contributions compared with our previous work are as follows:

- A more detailed explanation of the concept of Spider.
- Discussion about the considerations to apply Spider to real-world packet processing, including IPv6 support.
- Intensive evaluations of Spider and recent software LPM methods from multiple perspectives.

The evaluation shows that Spider achieves major improvement (1.8–2.6 times for IPv4, and 2.2–3.2 times for IPv6) compared with the state-of-the-art methods and delivers the processing capacity of 34 ports of 100 Gbps interface with 16 CPU cores. The performance improvement of Spider opens up the possibility of packet processing at the terabit-class rate by software.

II. RELATED WORK

Maximizing the performance of LPM has been one of the most fundamental research topics in the IP networking area. LPM methods have been studied for both hardware and

software approaches because their application fields are different. For example, using hardware routers has been a common way to achieve traffic engineering and ensure Quality of Service (QoS) with feature-rich protocols such as MPLS [15], [16] in career networks for a long time. On the other hand, the software middleboxes and their platforms have been actively developed [17]–[22], and deployed in new areas where the operators focus on cost-effectiveness and development speed, such as the NFV infrastructure [1], [2] in data center [23] and mobile networks [24].

For LPM in hardware, TCAM is the most popular technology that provides LPM with predictable latency [25]–[27]. However, TCAM has drawbacks on heat, power consumption, and monetary cost [28]. FPGA is another way to achieve hardware LPM at lower development cost than TCAM [29]. Bando *et al.* proposed FlashTrie [30] as an LPM architecture that is applicable to FPGA, and it supports two million routes for IPv4 at the lookup rate of 200 Mlps in the paper. Hamid *et al.* applied the existing hash-based LPM algorithm [31] to FPGA [32], which achieves 263 Mlps. Theoretically, the rate of lookup per second indicates the capacity to support the same rate of packet processing per second.

For LPM in software, which we focus on, CPU-based methods have been well studied from the past to the present. The direction toward faster LPM in the CPU is to minimize latency for each memory access and then improve the throughput as a result. A binary trie and path-compressed trie [33] are basic trie-based algorithms for LPM. As the demand for faster LPM methods increased along with the interface speed, optimized variants of trie-based algorithms were proposed [34]–[37]. Eatherton *et al.* proposed Tree Bitmap [38], which minimizes the latency of lookup by reducing the number of memory references. The key technique to reduce the number of memory references is to compress a multiway trie using a bitmap; this concept, which is to compress the data structure, has affected recent software LPM methods. Gupta *et al.* proposed DIR-24-8 [39], focusing on the fact that the ratio of /24 or shorter prefixes are the majority on routing tables of ISPs. DIR-24-8 has a data structure that achieves $O(1)$ lookup for each destination in most cases by expanding /24 or shorter prefixes into a single array. In addition, several studies proposed exploiting the Bloom filter to reduce the number of memory accesses tolerating false positives [40]–[42].

A common concept of modern software LPM methods is to exploit the access speed of the CPU cache to achieve shorter lookup latency [5]–[9]. As a result, they focus on how to minimize the data structure to fit the routing table into the CPU cache, as illustrated in the top row of Figure 2. We observed that most of their routing table fits in the L1 data CPU cache due to the increase of the cache size in the current CPU, although they initially intended to fit the routing table into the L3 CPU cache. DXR [11] proposed using a single large array where shorter length prefixes are directly expanded. DXR exploits the CPU cache by minimizing the number and range of memory accesses by its data structure

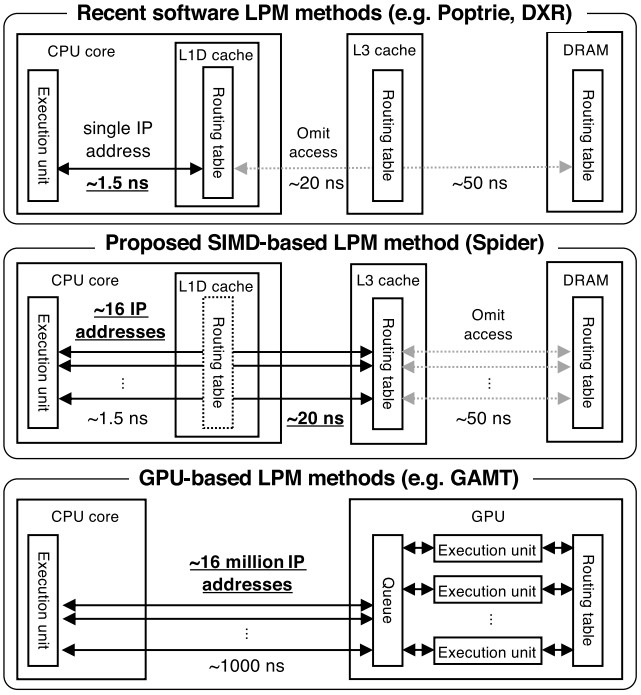

FIGURE 2. The architecture overview of recent LPM methods including Poptrie and DXR, GPU-based methods, and our LPM methods. The access latency and parallelism are reference values from the literature [43]–[45].

design, and it has achieved 115 Mlps for a single CPU core and random lookup in the paper. Poptrie [10] is also a method of this sort that inherits the concept of Tree Bitmap to reduce the latency of lookup. Poptrie uses a variant of a multiway trie that can omit the memory area for unnecessary child nodes to fit the multiway trie in the CPU cache. The key idea of Poptrie is efficiently to count the number of two types of child nodes by the population count instruction, to omit the memory area of either type of child node. As a result, Poptrie has achieved 240 Mlps with a random traffic pattern and a BGP full route as of 2015. Therefore, Poptrie overcomes the 100 Gbps wire-rate by a single CPU core, but at about 80% of the 200 Gbps wire-rate. Yang *et al.* have proposed SAIL [12], which takes the approach to split LPM into three levels to minimize the processing steps of its lookup procedure. In Poptrie's paper, SAIL achieves 159 Mlps, which is less than the lookup rate of Poptrie, although SAIL achieves a slightly faster rate than Poptrie with the BGP full route of 2013 [46].

Another concept of software LPM methods is to utilize discrete GPUs for LPM to maximize the lookup rate. Transferring data to GPUs involves additional latency due to communication over PCIe; therefore, GPU-based LPM methods conceal the latency by transferring many destination IP addresses at the same time and exploit massive parallelism on GPUs as illustrated in the bottom row of Figure 2. Han *et al.* proposed the architecture of a software router called PacketShader [47], which uses the GPU as a packet processing accelerator. PacketShader has achieved LPM on a GPU using DIR-24-8, while it has focused on not only

LPM but also the entire architecture of the software router. While PacketShader treats routing tables as static information, Zhao *et al.* proposed a GPU-accelerated LPM engine called GALE [48], which provides an incremental update on the GPU. Li *et al.* proposed a GPU-based LPM method called GAMT [45], which aims for both high performance and scalability. GAMT builds a routing table as a multibit trie and encodes it into a two-dimensional array, which is called the state-jump table. According to the optimization on the routing table, GAMT has achieved over 1,000 Mlps when 16 million destination IP addresses are processed at the same time. Moreover, several other GPU-based methods have been proposed [45], [49]–[51], which have a similar tendency to achieve high lookup rates with large batch sizes.

When considering LPM methods, it is easy to minimize the size of the routing table and the number of references to it with IPv4 because its address length is just 32-bit; however, IPv6 is not, although it is required in the practical packet processing software. Thus, some LPM methods do not support IPv6 by design [11], [48]. For example, DXR, which is a range-based LPM algorithm, does not support IPv6 by design, while focusing on the performance for IPv4 by exploiting the ratio of shorter prefixes. In addition, GALE, which is a GPU-based LPM method, does not support IPv6 because it depends on a large array on the GPU memory that stores all possible prefixes. Moreover, because the performance of LPM methods that rely on the shortness of the IPv4 address would degrade in IPv6 LPM, there is a gap between simply supporting the functionality of IPv6 LPM and achieving high performance with it. For example, DIR-24-8 can support IPv6, and its implementation has been available [21]; however, its performance would not be so high because its performance depends on the ratio of the shorter prefixes that fit in a single large array. Moreover, SAIL proposes to combine its method and a Bloom filter to support the large IPv6 routing table because its performance would degrade along with the length of the destination prefixes. Some other trie-based algorithms such as Poptrie support IPv6; however, the methods have not shown a clear solution for performance scalability against the longer address length and larger routing table of IPv6 LPM.

As this section showed, software LPM has been well studied; nevertheless, practically achieving the LPM performance over 100 Gbps is still challenging. The LPM methods in CPUs have minimized latency for a single lookup by exploiting the CPU cache. However, the performance of Poptrie, which is the state-of-the-art method in that approach, is insufficient for accommodating 200 Gbps by a single CPU core. Because the memory reference part of its lookup procedure has already been optimized, its further performance improvement depends on the increase of the CPU frequency, which has stagnated. Moreover, applying the LPM methods that rely on the CPU cache to IPv6 is harder than IPv4 because IPv6 LPM requires more memory space for its routing table, which is too large to fit it in the CPU cache. On the other hand, the performance of GPU-based methods depends on allowable

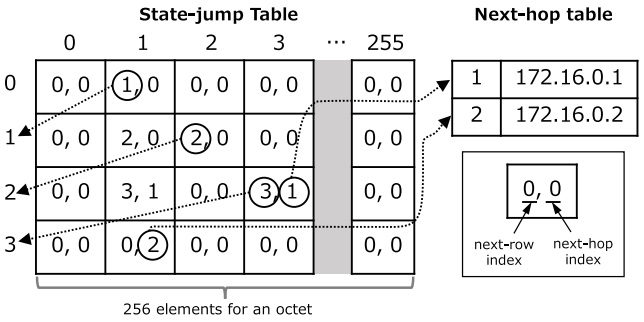

FIGURE 3. An example of a state-jump table and its next-hop table: 1.2.3.0/24 and 1.2.3.1/32 are installed, and their next-hops are 172.16.0.1 and 172.16.0.2.

lookup latency, because a large number of IP addresses must be prepared in advance to conceal the latency of communication over PCIe. For example, the lookup rate of GAMT declines to 50 Mlps when the prepared number of destination IP addresses is less than 4,000. To achieve even faster LPM practically, we propose yet another approach, which is to parallelize LPM in a single CPU core.

III. ARCHITECTURE

We propose a novel LPM method called `Spider`, which improves the lookup rate of LPM by parallelizing its lookup procedure inside a single CPU core. Spider leverages SIMD instructions for the parallelization instead of fitting the routing table into the higher-level CPU cache, which is the approach of the previous works. In Spider, a single CPU core processes up to 16 destination IP addresses, which correspond with each SIMD way one by one, at the same time reducing the latency for referencing to its routing table to around 20 nanoseconds because its routing table mostly fits in the L3 CPU cache, as illustrated in the middle row of Figure 2. Its degree of parallelism is significantly less than the GPU-based methods; however, Spider does not involve a longer access latency over PCIe.

The routing table structure of Spider is carefully designed so that the whole LPM procedure is composed of the SIMD instructions. The routing table design avoids the following three operations unsuitable for the SIMD mechanism: bitwise operation, pointer referencing, and conditional branching. Without these operations, it is difficult to compress the routing table to fit it into the higher-level CPU cache as with previous methods; instead, by designing the routing table to be processed without these operations, the SIMD instructions can efficiently process the routing table in parallel. This design choice is also suitable to achieve faster IPv6 LPM in addition to IPv4 because IPv6 LPM requires more memory space than IPv4. This section describes the design (§ III-A) of the routing table structure of Spider, the detailed lookup procedure with the SIMD instructions (§ III-B), and the management way of route information (§ III-C).

A. ROUTING TABLE DESIGN

We employ a state-jump table [45] as the data structure to process LPM avoiding the operations that are inefficient with

SIMD instructions. A state-jump table is a two-dimensional array expression of a routing table, which achieves LPM with just tracing elements on the array according to the target IP address; the mechanism avoids pointer referencing, which is unsuitable for SIMD instructions, in the lookup procedure. In addition, we employ two contrivances to avoid bit-wise operations in the lookup procedure: the byte-aligned data length, and the fixed-length stride of eight. These two conditions enable processing LPM using only byte-aligned operations in the lookup procedure.

Figure 3 shows an example of a state-jump table and its next-hop table of Spider. Each row has 256 elements, the number of patterns that an 8-bit field can represent; thus, the column in the state-jump table corresponds with each octet of the IP address. Besides, each element in a row contains two 2-byte fields: next-hop index, which represents the index of the next-hop IP address in a next-hop table, and next-row index, which represents the index of a row corresponding with the next octet of the target IP address. If the target IP address to look up is 1.2.3.1, its LPM procedure starts from looking at the element at (0, 1). Because the element's next-row index is 1 and the next octet of the IP address is 2, the LPM procedure then looks at the element at (1, 2). Similarly, the LPM procedure will look at the elements at (2, 3) and (3, 1) one after another, and the next-hop index obtained at last, which is 2 in this case, is the result of LPM.

In the state-jump table of Spider, we prepare a special row, which is called End-of-Lookup (EoL), to avoid conditional branching for synchronizing the completion timing between the SIMD ways. After a SIMD way finishes its lookup procedure, the SIMD way loops on the EoL until all SIMD ways finish their LPM procedure. To achieve such a mechanism, the EoL points itself by having all values zero in its elements. In addition, the elements that have no descendent row in the state-jump table automatically point to EoL because EoL is assigned the 0th row of the state-jump table, and elements have zero as the next-row index by default. In other words, the EoL is a sentinel in the routing table of Spider.

In addition, we minimize the amount of searching on the routing table by employing an optimization called direct pointing [10]. Consequently, the routing table of Spider consists of two parts: direct pointing table and state-jump table. In direct pointing, an array corresponding to the part from the beginning of the IP address is prepared in advance, and the lookup for the length is covered with a single access to the array. The optimization would reduce CPU cycles in the lookup procedure of Spider by omitting the access to the state-jump table. In Spider, the direct pointing table has 65536 elements, which corresponds with all possible values that a 16-bit length field takes so that the direct pointing table can cover the lookup for the first two octets of the IP address. For the length covered with the direct pointing table, 16-bit is suitable for Spider. Most of the SIMD instructions can address only byte-aligned values, and the size of the direct pointing table becomes too large to fit in the last level cache in the CPU with more than 16-bit length.

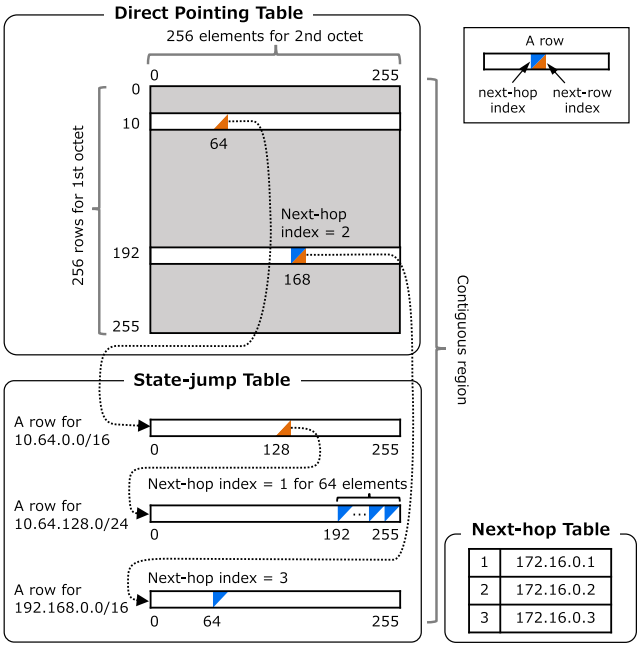

FIGURE 4. Design of Spider's data structure: an example of a routing table in which 10.64.128.192/26 via 172.16.0.1, 192.168.0.0/16 via 172.16.0.2, and 192.168.64.0/24 via 172.16.0.3 are installed, and 192.168.64.1 is processed.

Figure 4 illustrates an example of the routing table of Spider employing direct pointing. In this example, three routes (10.64.128.192/26 via 172.16.0.1, 192.168.0.0/16 via 172.16.0.2, and 192.168.64.0/24 via 172.16.0.3) are installed. In the routing table, the lookup for 192.168.64.1 is processed in two phases: lookup with the direct pointing table and that with the state-jump table. First, the lookup with the direct pointing table is processed. The lookup procedure loads the element corresponding to the first and second octets of the IP address from the direct pointing table. In this example, because the element contains a next-hop index for 172.16.0.2, the lookup procedure saves the next-hop index as a result. In addition, because the element also contains a next-row index for the row representing 192.168.0.0/16, the lookup procedure will trace the row to search for the longer result. Then, because the processed length on the destination IP address exceeds two octets, which a direct pointing table can cover, the lookup procedure enters the lookup with the state-jump table. The lookup procedure loads the element corresponding to the third octet of the IP address from the row representing 192.168.0.0/16. At this time, because the 64th element contains a next-hop index for 172.16.0.3, the lookup procedure overwrites the result with the next-hop index. The next-hop index will be the longest matched result because the following procedure will not find a new result in this case. Finally, because the 64th element of the row for 192.168.0.0/16 does not contain the next-row index, the lookup procedure detects the completion of the lookup.

B. PARALLELIZATION WITH SIMD INSTRUCTIONS

Based on the routing table described in Section III-A, Spider performs LPM at a higher lookup rate by looking up

Algorithm 1 Lookup Procedure for IPv4

Input: *DstArray*
Output: *ResArray*

```
1:  load256(dst, DstArray);
2:  /* Direct pointing for first two octets of IPs */
3:  idx = shuffle8(dst, maskd16);
4:  idx = add32(idx, 256); // row[1] + idx
5:  val = gather32(fib, idx);
6:  nhi = shuffle8(val, masknhi);
7:  res = nhi;
8:  nri = shuffle8(val, masknri);
9:  while not all next-row indexes are 0 do
10:     /* Iterative lookup for subsequent octets of IPs */
11:     idx = shuffle8(dst, maskd8);
12:     idx = add32(idx, nri); // row[next-row index] + idx
13:     val = gather32(fib, idx);
14:     nhi = shuffle8(val, masknhi);
15:     maskbl = cmpeq32(nhi, 0);
16:     res = blend32(maskbl, res, nhi);
17:     nri = shuffle8(val, masknri);
18: end while
19: store256(ResArray, res);
20: return;
```

multiple IP addresses in parallel with SIMD instructions. The lookup procedure basically iterates two parts of processing: data loading and manipulation. Both parts consist of only SIMD instructions, which can process the data in parallel, but cannot address bit-wise operations, pointer referencing, and conditional branching without extra operations; these characteristics are the reason we avoided these operations when designing the routing table of Spider. For the data loading, Spider loads multiple elements onto a SIMD register in parallel from the routing table with the `gather` instruction. The `gather` instruction can load multiple data located in separated regions; a novelty of Spider is to focus on the `gather` instruction enabling to compose the lookup procedure only with SIMD instructions. For the data manipulation, Spider calculates the locations of the elements to search longer matching results with `shuffle` and `add` instructions. In the lookup phase, we assume that the SIMD instructions can process eight 32-bit data in parallel, which correspond with 8 IP addresses one by one, referencing Intel's AVX2 mechanism [52].

Algorithm 1 describes the lookup procedure of Spider at the instruction level. The subscripts of each instruction represent the length of data type that the SIMD instruction operates on. The lookup procedure consists of two phases: looking up the direct pointing table for the first two octets of the target IP address and iterative processing for subsequent octets of the target IP address. First, the procedure processes the first phase for looking up a direct pointing table, which consists of the following four parts. The first part is the extraction of the first two octets from the IP addresses using a `shuffle` instruction (line 3). The second part is the addition of the extracted parts to 256, which is the offset for the EoL, to calculate the indexes of the elements in the direct pointing table using an `add` instruction (line 4). The third part is the loading of the elements based on the indexes using the `gather` instruction (line 5). The fourth part is the extraction of the next-hop index and next-row index from the elements using the `shuffle` instruction (lines 6 and 8). The lookup procedure finishes when all next-row indexes indicate zero; in this case, if all longest matched results of the IP addresses are /16 or shorter, the lookup procedure finishes. Otherwise, then the lookup procedure enters the second phase, which processes subsequent octets by iteration for each octet. The second phase is similar to the first phase but has two different points. The first point is that the single octet is extracted using the `shuffle` instruction per iteration (line 11). The second point is that the next-row index extracted in the previous phase or iteration is used to calculate the locations of the elements in the state-jump table (line 12). When next-row indexes of all IP addresses become zero, the lookup procedure finishes, and the last obtained next-hop indexes are the longest matched results for the target IP addresses.

Moreover, we maximize the lookup rate of Spider by introducing an optimization technique called loop fusion [53]. Loop fusion is a common optimization to increase the throughput of an iterative procedure by combining two independent iterations into a single loop to conceal the latency of memory access. In the lookup procedure of Spider, the SIMD instructions that cause memory referencing such as load, store, and gather are frequently used. By processing another iteration instead of just waiting for the completion of memory referencing instructions, the throughput of the lookup procedure increases. Because Spider combines two iterations that consist of 8-way SIMD instructions, the resulting procedure processes 16 IP addresses in a single iteration.

C. ROUTING TABLE MANAGEMENT

Software LPM methods need to manage route information in their routing tables while keeping a higher lookup rate in the lookup phase. Spider also must have the basic feature to manage route information such as addition and deletion. However, the state-jump table, which Spider employs, is unsuitable for these operations, because it cannot represent route information other than best routes with contiguously arranged elements that are minimized to store only two indexes required to represent best routes. Without the route information other than best routes, when an old best route that overlaps the next highest priority route is deleted, the new best route cannot be activated. On the other hand, to achieve a higher lookup rate in the lookup phase, arranging elements contiguously and minimizing their size are required for improving LPM performance. Otherwise, the CPU cache hit rate, especially for the traffic pattern with high locality of destination IP addresses, would decrease, which is not negligible even though Spider does not prioritize to exploit the access speed of the CPU cache. Consequently, the state-jump

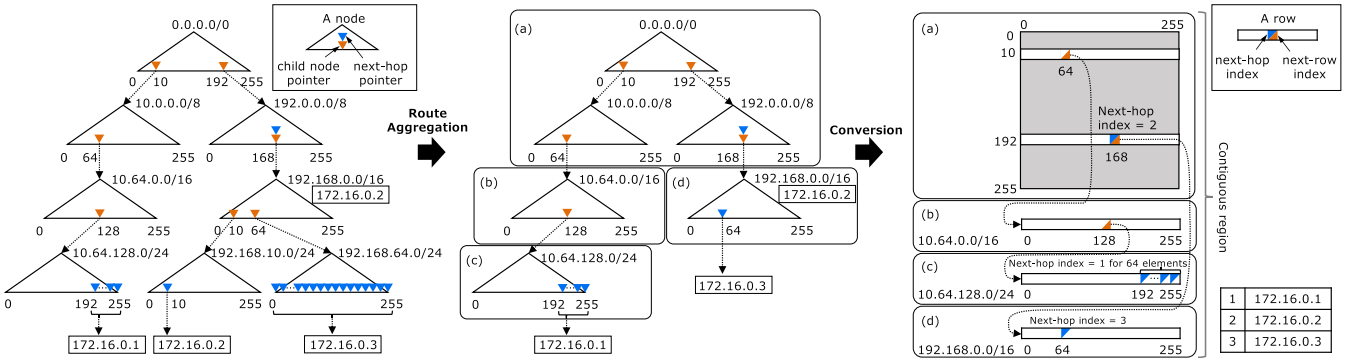

FIGURE 5. Spider maintains a multiway trie that route aggregation is applied to, and converts the multiway trie into the real routing table that consists of a direct pointing table and state-jump table. In this example, the routes for 10.64.128.192/26 via 172.16.0.1, 192.168.0.0/16 via 172.16.0.2, 192.168.10.10/32 via 172.16.0.2, and 192.168.64.0-255/32 via 172.16.0.3 are installed.

table achieves efficient processing with SIMD instructions, but it cannot achieve route addition and deletion without performance degradation in the lookup phase.

To achieve both route management and higher lookup rate, Spider maintains route information on a multiway trie for only management and uses the state-jump table converted from the multiway trie in the lookup procedure. The multiway trie is an extended variant of the trie data structure, where each node holds 2^k descendent nodes, while in the normal trie, each node holds only two descendent nodes. Thus, the multiway trie can provide basic route management features as with the normal trie. In addition, the use of a 2^8 multiway trie can simplify the conversion procedure to the state-jump table because a node in the multiway trie can be converted to a row for the state-jump table without transforming the structure of the tree. Therefore, the characteristic provides faster build of the state-jump table, which is an advantage to support frequent route updates in real-world environments. On the other hand, a disadvantage of the multiway trie is memory space inefficiency compared with the normal trie. However, the disadvantage is negligible in Spider because Spider uses only a state-jump table converted from a multiway trie in its lookup procedure.

In real packet processing, there are two ways to update a routing table: batch-based and incremental updating. Spider uses batch-based updating because batch-based updating can practically coexist with the routing table structure of Spider. By contrast, applying incremental updating to Spider requires route management features such as route addition and deletion on the state-jump table, which would lead to performance degradation of the lookup procedure as described before. In batch-based updating, the software holds route updates for certain seconds, and after the holding period, reconstructs its routing table entirely. The batch-based updating is employed in certain career-grade hardware routers [54] because it can handle frequent route updates without supporting the incremental updating feature on the FIB table, which prioritizes performance. The design contributes to simplifying the implementation and improving the consequent performance of the FIB table. Similarly, in Spider, the batch-based updating can handle frequent route updates without performance degradation in the lookup procedure due to the complication of the state-jump table. Because Spider's state-jump table has a role similar to the FIB table, it is natural that the batch-based updating in Spider brings out a similar advantage with the hardware routers. In addition, batch-based updating can achieve synchronization in a multithread environment by simply replacing the old routing table with the new one by Read Copy Update [55], while incremental updating requires fine-grained exclusive control for synchronization between the threads for each operation that changes the routing table.

In addition, we maximize the lookup rate by reducing the size of the resulting routing table with an optimization called route aggregation [56]. Route aggregation reduces the number of routes in a routing table by combining a plurality of routes that have the same next-hop into one route. The reduction in the number of routes leads to a reduction of the size of the routing table, which increases the lookup rate because of its higher data locality. In Spider, the optimization is applied to the nodes that do not affect the longest matching result, such as the nodes that have the same next-hop for all elements.

Figure 5 shows the process of conversion from the original multiway trie to the real routing table, which consists of two processes. First, the route aggregation is applied to the multiway trie. The node for 192.168.10.0/24 is omitted because its elements do not affect the lookup result; the next-hop of the element, which is 172.16.0.2, is the same with the next-hop for 192.168.0.0/16. In addition, the node for 192.168.64.0/24 is omitted because all its elements have the same next-hop, which is 172.16.0.3; all next-hop from the elements are aggregated into the corresponding element in the node for 192.168.0.0/16. Second, the multiway trie is converted to a real routing table by the following three processing parts. The first part is the arrangement of all /8 or shorter routes information in the original multiway trie into a single array of 65536 elements to build the direct pointing table. The second part is the conversion of each node of the

multiway trie that represents /16 or longer routes into a single array that has 256 elements, which will be each row in the resulting state-jump table. The third part is the arrangement of the direct pointing table and all rows for the state-jump table into a two-dimensional array. In this example, each region on the aggregated multiway trie (a–d) corresponds to each region with the same symbol in the consequent routing table. The consequent routing table becomes the same as the routing table described in Section III-A. For each element in a row, Spider calculates two indexes. One is the next-row index, which is a row number in the state-jump table converted from each memory address pointer to the descendent nodes. Another is the next-hop index, which is the index number of the next-hop IP address in the next-hop table. Because the number of rows and next-hops has to be known before the conversion, Spider maintains the number along with the original multiway trie.

IV. APPLICABILITY TO REAL PACKET PROCESSING

In this section, we discuss considerations when applying Spider to real-world packet processing applications. Because real-world packet processing applications require to support IPv6, we consider applying Spider to IPv6 LPM. Spider does not intend to fit the routing table size into the CPU cache; therefore, this design choice enables adapting Spider for IPv6, which requires more memory space for the routing table than IPv4. In addition, we discuss the effect of Spider on the latency in real packet processing. Spider processes up to 16 destination IP addresses on a single lookup procedure. This parallelism is required to maximize the performance; however, it may cause a latency increase for forwarding the packets.

A. IPv6 APPLICABILITY

Recent packet processing software has to support IPv6 because it performs an aspect of the Internet. In fact, the growth rate of IPv6 has far exceeded IPv4, and the number of IPv6 BGP routes has reached 9.7% of IPv4 as of 2019 [57], with the background of IPv4 exhaustion. The large address and the growing number of routes cause the increase of routing table size, which leads to the difficulty in fitting it into the CPU cache—especially the L1 data CPU cache. In contrast, Spider improves the lookup rate of the LPM by the parallelism instead of only exploiting the CPU cache. Therefore, Spider can derive an improvement on the larger routing table for IPv6.

The lookup procedure of Spider can easily adapt to IPv6 with some additional operations to handle the address length of IPv6, which is four times as long as IPv4. Despite the additions, Spider achieves up to 16 parallel LPM for IPv6 as with IPv4. Algorithm 2 describes the IPv6 lookup procedure of Spider at the instruction level. The procedure is basically the same as with the lookup procedure for IPv4 except for the parts to handle the longer address length. In the lookup procedure for IPv6, we take the approach that prioritizes the parallelism of the lookup procedure over the overhead for refilling each data on the SIMD register per four octets from the original IPv6 addresses.

Algorithm 2 Lookup Procedure for IPv6

Input: *DstArray*
Output: *ResArray*

1: $load_{256}(dst1, DstArray[0])$;
2: $load_{256}(dst2, DstArray[2])$;
3: $load_{256}(dst3, DstArray[4])$;
4: $load_{256}(dst4, DstArray[6])$;
5: /* Extract first four octets from IPv6 addresses */
6: $unpack1 = unpacklo_{32}(dst1, dst3)$;
7: $unpack2 = unpacklo_{32}(dst2, dst4)$;
8: $dst = unpacklo_{32}(unpack1, unpack2)$;
9: /* Direct pointing for first two octets of IPs */
10: $idx = shuffle_8(dst, maskd16)$;
11: $idx = add_{32}(idx, 256)$; // row[1] + idx
12: $val = gather_{32}(fib, idx)$;
13: $nhi = shuffle_8(val, masknhi)$;
14: $res = nhi$;
15: $nri = shuffle_8(val, masknri)$;
16: **while** not all next-row indexes are 0 **do**
17: /* Iterative lookup for subsequent octets of IPs */
18: $idx = shuffle_8(dst, maskd8)$;
19: $idx = add_{32}(idx, nri)$; // *row[next-row index] + idx*
20: $val = gather_{32}(fib, idx)$;
21: $nhi = shuffle_8(val, masknhi)$;
22: $maskbl = cmpeq_{32}(nhi, 0)$;
23: $res = blend_{32}(maskbl, res, nhi)$;
24: $nri = shuffle_8(val, masknri)$;
25: **if** 4 octets has been processed **then**
26: /* Left-shift by four octets and
27: extract the beginning four octets */
28: $dst1 = srli_{256}(dst1, 4)$;
29: $dst2 = srli_{256}(dst2, 4)$;
30: $dst3 = srli_{256}(dst3, 4)$;
31: $dst4 = srli_{256}(dst4, 4)$;
32: $unpack1 = unpacklo_{32}(dst1, dst3)$;
33: $unpack2 = unpacklo_{32}(dst2, dst4)$;
34: $dst = unpacklo_{32}(unpack1, unpack2)$;
35: **end if**
36: **end while**
37: /* Fix order of data scrambled by unpack */
38: $res = permute_{32}(res, maskres)$;
39: $store_{256}(ResArray, res)$;
40: *return*;

Compared with the lookup procedure for IPv4, the following four processing parts are additionally required to support IPv6 LPM. The first part is the four times loading of IPv6 addresses from the input array by the `load` instructions due to the address length (lines 1-4). The second part is the extraction of the first four octets from the loaded IPv6 addresses by `unpacklo` instructions (lines 6-8). The `unpacklo` instruction interleaves two registers, and it also can be applied to data

extraction by repeating that operation. The reason why we adopted `unpacklo` for this extraction is that the three times execution of `unpacklo` marks the highest lookup rate in our microbenchmark compared with other ways such as the combination of bit-shift and bit-mask. The third part is the refilling of the target data with the next four octets from the rest of the IPv6 addresses by `srli` and `unpacklo` instructions when the lookup procedure reaches the four octets boundary of target IPv6 addresses (lines 25-34). The fourth part is the correction of the order of the results, which has been changed by `unpacklo` instructions, by the `permute` instruction to return results in the original order (line 37). The `permute` instruction can execute shuffling of the data overcoming the 128-bit boundary, while the `shuffle` instruction cannot.

B. INCREASED LATENCY BY PARALLELIZATION

To execute LPM in parallel, the method needs to wait for the arrival of the corresponding number of packets, and consequently, it might increase latency for forwarding the packets. Despite the requirement, the increase of latency for each packet would be insignificant in practical applications because the applications already exploit the optimization called packet batching [3], [20], [22], [47], which processes multiple packets at once to increase the throughput. The applications can easily introduce Spider without a significant increase of the latency for each packet because batching in the packet processing and parallelizing LPM are common in the necessity to wait for the arrival of multiple packets.

V. EVALUATION

For the evaluation of Spider, we consider the following three perspectives: the effect of the parallelism provided by the SIMD mechanism, performance comparison with other methods, and performance scalability. For the first perspective, we measured the performance changes along with the parallelism provided by the SIMD mechanism to show that the parallelization contributes to the performance of Spider (§ V-B). In addition, we measured the packet forwarding performance according to different batch sizes to clarify the allowable degree of parallelism in real-world applications (§ V-C). For the second perspective, we compare the lookup rate and CPU cycles per iteration of Spider with other methods to show that Spider outperforms other methods in terms of the lookup rate while showing longer latency for an iteration (§ V-D). For the third perspective, we measured the changes of lookup rate along with the CPU frequency (§ V-E) and the number of CPU cores (§ V-F) to clarify the performance scalability of Spider under various conditions.

For the methods except for Spider, we used the publicly available implementations of DXR [17], [58], Poptrie [59], and an implementation of DIR-24-8 [39] from DPDK [21] with modifications to measure the performance. Both DXR and Poptrie vary regarding the length of direct pointing, whose length is represented as the name, such as Poptrie_{18} and D18R. The expression manners are aligned with the original papers of each method [10], [11]. For Spider, we represent the settings of parallelism as Spider_{16w}. The equipment used for experiments consisted of an Intel Xeon Gold 6130 CPU (2.10 GHz, 3.70 GHz with turbo-boost, 22 MiB cache, 16 cores) and 48 GB DDR4-2666 memory. The sizes of the L1, L2, and L3 CPU caches are 1 MiB, 16 MiB, and 22 MiB, respectively.

A. DATASET AND TRAFFIC PATTERNS

We evaluate the performance of the LPM methods with the current BGP route of the Internet, which is called the BGP full route. BGP full route is suitable for this evaluation because IP routing based on the BGP full route is the highest load situation that current routers face in real-world environments. As the current BGP routes of the Internet, we used two BGP full routes of real ISPs: ISP-A and ISP-B. For IPv4, the numbers of the routes were 773, 822, and 776, 684, and the ratio of /24 prefix was 58% for both datasets. The datasets were captured on December 10, 2019. For IPv6, the numbers of the routes were 77, 674 and 78, 156, and the ratio of /32 prefix was 17% for both datasets. The datasets were captured on March 4, 2020.

We consider the random and real-trace traffic patterns for the evaluation in this paper. The random traffic pattern reveals the worst-case performance of LPM methods, and the real-trace traffic pattern reveals the performance against actual traffic. For the random traffic pattern, we measured the time of looking up 2^{32} random destination IP addresses with a just-in-time generation of the pattern aligned with Poptrie's paper. We used the linear congruential method to generate a random traffic pattern. As a problem peculiar to IPv6, the fully random traffic pattern would not show the worst-case performance of LPM methods, because most IPv6 addresses would not match any route due to its huge address space.

To evaluate the worst-case performance of IPv6 LPM methods, we made a situation where most addresses match some random routes by following three processes. First, a pattern table that includes the first 32-bit of the prefixes in the target routing table is generated in advance of the measurement. Second, a 32-bit pattern is picked randomly from the table. Third, a 128-bit IPv6 address is generated by concatenating the 32-bit pattern picked from the table, 32-bit random part, and 64-bit host part, which is fixed to ::1. For the real-trace traffic pattern, we measured the time of looking up the destination IP addresses from real Internet traffic captured on April 10, 2019, on the sampling point F of the WIDE backbone, which is published as the MAWI dataset [60]. For this pattern, the 2^{28} IP addresses for IPv4, and the 2^{26} IP addresses for IPv6 are arranged in a 1-Gbyte array in advance of the execution, as the maximum number that can be prepared without affecting the lookup performance.

B. EFFECT OF PARALLELIZATION IN SPIDER

Because maximizing the lookup rate by parallelizing LPM is our main contribution, we evaluated how the increase of parallelism by the SIMD instructions contributes to the

TABLE 1. Memory footprint of routing tables.

	ISP-A	ISP-B
Method	Memory size [MiB]	Memory size [MiB]
Poptrie_{18}	1.25	1.35
D18R	1.29	1.40
Spider_{16w}	9.32	11.61
DIR-24-8	64.15	64.34

performance. The measurement was conducted with the random traffic pattern and the BGP full route of both ISPs.

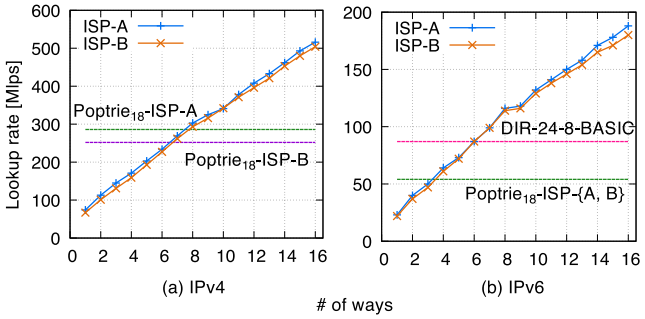

FIGURE 6. Lookup rate scales with the parallelism by the SIMD instructions in Spider.

According to the result shown in Figure 6, we confirmed the lookup rate of LPM scales along with the parallelism provided by the SIMD mechanism. The result indicates that the parallelization with SIMD instructions contributes to the improvement of LPM performance. When the degree of parallelism is 8-way or more for IPv4, and 7-way or more for IPv6, the lookup rate by Spider outperforms other methods that take approaches to minimize memory access latency. For IPv4, when the degree of parallelism is 16-way, Spider reaches 516 Mlps for ISP-A, which is 6.9 times faster than the case when the parallelism is 1-way, 1.8 times faster than Poptrie_{18}. For IPv6, when the degree of parallelism is 16-way, Spider reaches 188 Mlps of lookup rate for ISP-A, which is 8.2 times faster than the case when the parallelism is 1-way, 2.2 times faster than DIR-24-8. Note that the sublinear scaling between 8-way and 9-way is due to the load of random address generation by the just-in-time manner. More than 9-way needs two times the generation process, and the process for IPv6 is heavier than IPv4. Therefore, a performance stagnation between 8-way and 9-way is clearly observed in IPv6.

When the parallelism of Spider is no more than 6-way for IPv4, the lookup rate of Spider falls below the others. The reason Spider cannot outperform other methods in lower parallelism is the drawbacks for using SIMD instructions, which are the larger memory footprint and the reduction of the CPU frequency. First, the larger memory footprint leads to the reduction of the CPU cache hit rate, which leads to longer access latency in the lookup procedure. For Spider, as shown in Table 1, the memory footprint is larger than the 1 MiB of the L1 data CPU cache and Poptrie_{18} and D18R, although it would fit in the 22 MiB of the L3 CPU cache. Therefore, Spider's size of memory footprint leads to longer access latency than Poptrie_{18} and D18R in the lookup procedure. Second, the reduction of the CPU frequency, which is caused by the power consumption increase for SIMD processing, leads to the latency increase of each lookup. The frequency reduction reaches 19% for the CPU used for this evaluation, according to the specification [61]. In Spider, when the improvement of the lookup rate by the parallelism of SIMD instructions does not overcome these drawbacks, the performance becomes lower than for other methods.

C. DEGREE OF PARALLELISM ON REAL PACKET PROCESSING

To exploit the parallelism of Spider, which is confirmed to contribute to the lookup performance in Section V-B, we need to clarify the allowable degree of parallelism based on the feasible batch size that the packet processing software can use. As we described in Section IV-B, the degree of parallelism that can be increased without additional latency depends on the batch size in real packet processing. Therefore, the batch size to saturate the capacity of the recent NIC would be the allowable degree of parallelism at the same time. To measure the batch size in real packet processing, we implemented an IP-based packet forwarding application called `lpmtest`, which is modulable for LPM methods, and measured its performance for both Spider_{16w} and Poptrie_{18}. The reason we measured the performance for Poptrie_{18} in addition to Spider_{16w} is to confirm the practical batch size without the effect of parallelizing LPM because Poptrie_{18} processes a single IP address per lookup in contrast to Spider_{16w}. Thus, the result of Spider_{16w} is complementary in this measurement, which shows that the characteristics of Spider_{16w} would not affect the appropriate batch size.

The design of `lpmtest` is the kernel-bypass architecture to bring out the performance, which is typical for high-speed packet processing software. In addition, the `lpmtest` does not have features and optimizations as in production-grade frameworks to eliminate the effects that do not relate to the scope of this measurement. The `lpmtest` directly operates on the Intel XL710 series NIC by a simple driver in user-space and processes IP-based forwarding for the packets by designated batch sizes. In multicore environments, the `lpmtest` can have multiple threads to process packets in parallel, using the RSS feature of the NIC. The threads are organized in the run-to-completion model, where each thread processes the entire forwarding process for assigned packets. The model is a major one in software that focuses on latency reduction. By contrast, the software focusing on maximizing throughput employs the pipeline model, where each thread is in charge of a single feature. The reason we adopted the run-to-completion model is that the model directly reflects the performance of LPM as its forwarding performance. On the other hand, the forwarding performance of pipeline model changes depending on the various factors, and thus the model is not suitable for measuring the performance of a single feature such as LPM. Moreover, the routing table in `lpmtest`

synchronizes with the kernel via NETLINK, so that we can insert routes using Linux standard utilities such as iproute2. Changes to the routing table are processed by batch-based updating described in Section III-C, which also proves the feasibility of the technique.

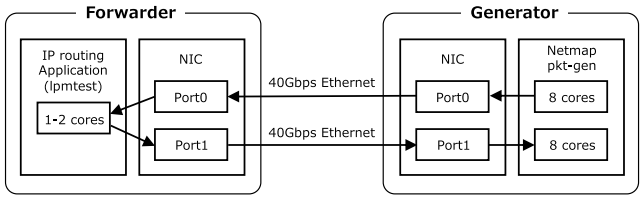

FIGURE 7. Setup for the IP forwarding test.

Figure 7 shows the setup of this measurement. In the setup, there are two nodes, which are generator and forwarder, and they are connected with the 40 Gbps Ethernet by Intel XL710 NICs. The measurement scenario consists of the following three processes. First, the generator sends the packets that have random destination IP addresses using a Netmap-based [18] packet generator using eight CPU cores out of 16. Second, the forwarder processes the packets by `lpmtest` under setup for each measurement. Third, the generator receives the packets using the eight CPU cores that are not used to send the packets. We confirmed that the number of packets from the generator overcomes the number that the forwarder can process in advance of the measurements. We adopted the 64-byte packet size because the size is the highest load situation for packet processing software. To clarify the performance bottleneck, we increased the number of CPU cores up to two as the maximum number at which the forwarding performance improved. The equipment of the generator and forwarder has the same specification described in Section V.

As a result, this measurement has revealed that the suitable batch size for IP-based packet processing is 16 or more when the capacity of the CPU is enough, and thus Spider's degree of parallelism of 16 is allowable for real-world packet processing. Figure 8 shows the packet processing rates of both Poptrie$_{18}$ and Spider$_{16w}$ with two types of routing tables: only with a default route and BGP full route. For Poptrie$_{18}$, when the number of CPU cores is two, the batch size at the upper limit of the packet processing rate is 16, and when the number of CPU cores is one, the batch size at the upper limit is 32, although the amount of increase from the batch size of 16 is insignificant. The results for Spider$_{16w}$ are mostly the same as the results of Poptrie$_{18}$. Moreover, from the results of Poptrie$_{18}$ and Spider$_{16w}$, the difference between default route only and BGP full route patterns is insignificant; therefore the results indicate that LPM is not the bottleneck in this measurement. Therefore, we can observe that the efficient batch size in real-world packet processing is 16 or more when the LPM is not the bottleneck, and thus the 16-way parallelism of Spider is suitable for recent packet processing software. Besides, the reason packet processing rate saturates

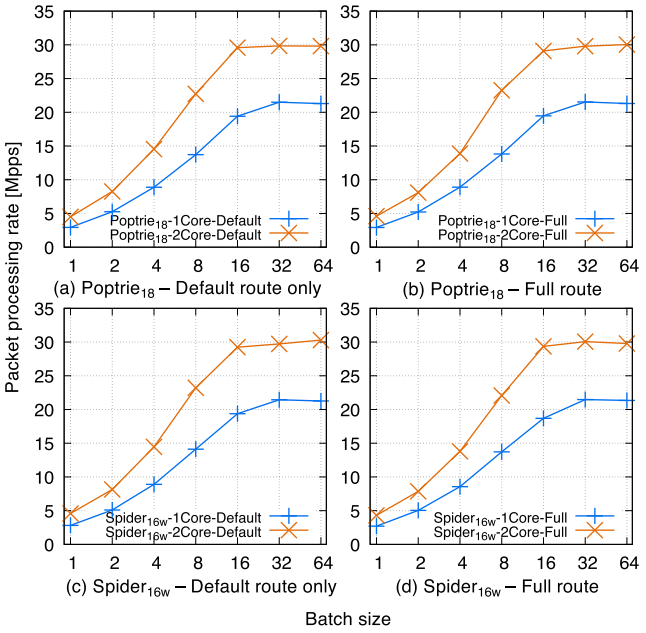

FIGURE 8. Packet processing rates according to different batch sizes.

around 30 Mpps is that there is a hardware performance limit over the packet I/O in the Intel XL710 series NIC, according to the benchmark by Intel [62].

D. COMPARISON OF LOOKUP RATE WITH OTHER METHODS

To show the performance advantage of Spider, we compare the lookup rate of Spider$_{16w}$ with Poptrie$_{18}$, Poptrie$_{16}$, D18R, D16R, and DIR-24-8 with random and real-trace traffic patterns and the BGP full routes of two real ISPs. For random lookup, we added 8-way and 1-way variants of Spider in addition to the original 16-way version to show the effect of parallelism provided by the SIMD mechanism, while we measured only the 16-way version of Spider for the real-trace lookup due to the experimental restriction of real-trace data.

Figure 9 shows a comparison of the lookup rate of the random and real-trace traffic patterns for IPv4 and IPv6. Through all experiments, Spider$_{16w}$ outperforms other methods for the BGP full routes of both ISP-A and ISP-B. The results show that parallelizing the lookup procedure with SIMD instructions can achieve higher lookup rates than other methods despite the larger memory footprint and the reduction of the CPU frequency. For IPv4, compared with Poptrie$_{18}$, which is the second-highest rate, Spider$_{16w}$ is 1.8 times faster in random lookup with the BGP full route of ISP-A and 2.0 times faster with ISP-B. In terms of real-trace lookup, Spider$_{16w}$ is 2.3 times faster than D18R, which is the second-highest rate, with the BGP full route of ISP-A and 2.6 times faster with ISP-B. For IPv6, compared with DIR-24-8, which is the second-highest rate, Spider$_{16w}$ is 2.2 times faster in random lookup with the BGP full route of ISP-A and ISP-B. In terms of real-trace lookup, the lookup rate of each method is significantly higher than that of the random traffic pattern. The

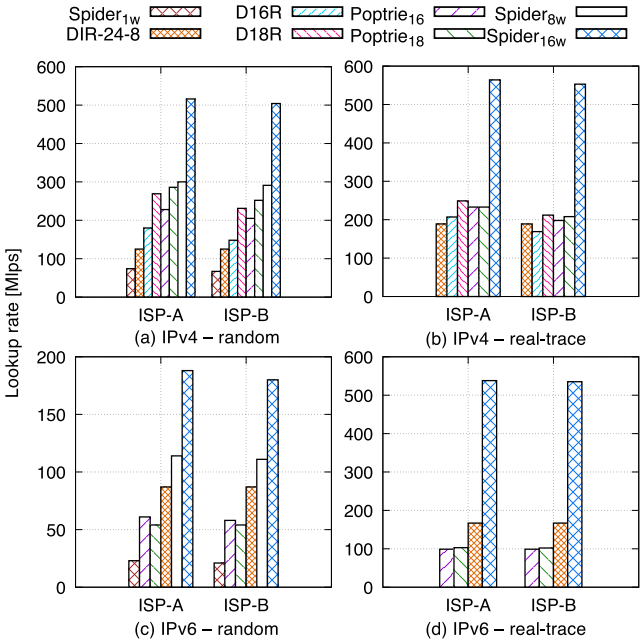

FIGURE 9. Comparison of the lookup rate of the random and real-trace traffic patterns for IPv4 and IPv6.

TABLE 2. The CPU cycles per iteration: Poptrie and DXR process one IP address per iteration, while Spider processes 16 IP addresses per iteration.

Method	Mean	50th	75th	90th	99th
Poptrie_{18}	60.69	58	58	60	172
D18R	54.24	52	52	54	186
Spider_{16w}	200.27	192	214	238	352

reason is the load to generate the random traffic pattern for IPv6 is higher than IPv4, although the just-in-time generation is required to measure the worst-case performance with the huge address space of IPv6. Moreover, the lookup rate of Spider in the real-trace traffic pattern is far higher than other methods for both IPv4 and IPv6. This is because Spider can utilize the L1 data CPU cache when the traffic pattern has high locality of destination IP addresses, in addition to the parallelism provided by the SIMD mechanism. As a result, Spider_{16w} is 3.2 times faster than DIR-24-8, which is the second-highest rate, with the BGP full routes of ISP-A and ISP-B.

In addition, we measured CPU cycles to process the lookup procedure for each method to reveal the latency characteristics. Figure 10 shows the CDF of the CPU cycles per iteration for each method, and Table 2 summarizes the mean, 50th (median), 75th, 95th, and 99th percentiles of the CDF. For the measurement manner, we measured the CPU cycles to process a single iteration, not a single IP lookup, because previous methods and Spider_{16w} are different in the number of IP addresses processed in a single iteration. Thus, the CPU cycle equals 1 IP lookup for Poptrie_{18} and D18R, while it equivalents 16 IP lookups for Spider_{16w}. As the overall trend of the result, Spider_{16w} requires four times or more CPU

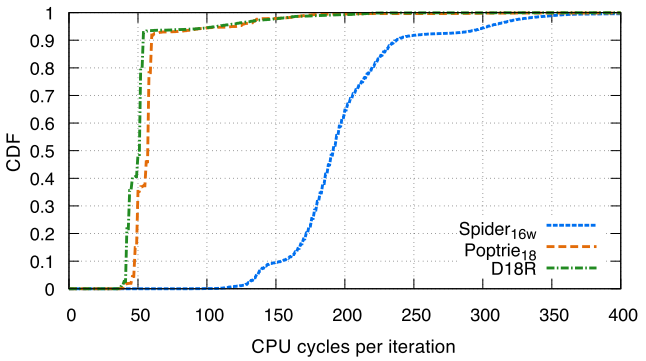

FIGURE 10. CDF of the CPU cycles per iteration.

cycles than other methods; this arises from the longer memory access time to process 16 IP addresses such as gathering data in the lookup procedure. In addition, the result in Section V-C indicates that the latency would not affect the performance of real-world applications because of the packet batching. For Poptrie_{18} and D18R, the trend of the result is almost the same, although D18R slightly outperforms Poptrie_{18}. We can observe the small degradation of the CPU cycles in Poptrie_{18} and D18R when CDF is around 0.4, which is caused by the miss-hit in the L1 data CPU cache. However, the performance degradation until CDF reaches 0.9 or more is small; it indicates they fit most of their routing table into the L1 data CPU cache.

E. LOOKUP PERFORMANCE ACCORDING TO CPU FREQUENCY

To confirm that the performance advantage of Spider remains at all CPU frequencies, we measured the performance of LPM methods under different CPU frequencies. Through this measurement, we can observe two aspects of the performance of each LPM method: the performance at lower CPU frequencies indicates the possible advantage of the devices that have limited computing resources such as Internet-of-Things (IoT) nodes and Customer–Premises–Equipment (CPE), and the performance under higher CPU frequencies indicates the possible performance scalability with future CPU advances. The measurements were conducted for the random traffic pattern using `acpi-cpufreq` as the CPU frequency driver, which also disables the turbo-boost feature in Intel's CPU. Both the lowest CPU frequency of 1.0 GHz and the highest CPU frequency of 2.1 GHz are the hardware limits. Although the CPU used in this experiment reduces its frequency dynamically when using SIMD instructions, the reduction range is insignificant when using a single CPU core; thus, we plot the performance of Spider at the configured CPU frequency on figures in this section and Figure 1 in Section I.

Figure 11 shows the performance of LPM methods at different CPU frequencies for IPv4 and IPv6. Through both results, we can observe that Spider_{16w} has an advantage not only at higher CPU frequencies but also at lower CPU frequencies, because the parallelism, which is a factor of

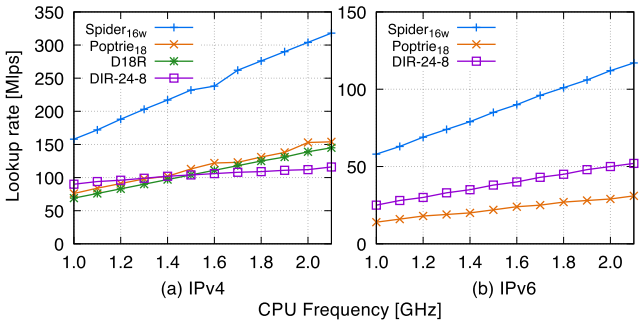

FIGURE 11. Performance of LPM methods under different CPU frequencies for IPv4 and IPv6.

Spider's performance advantage, is not affected by the CPU frequency. In addition, Spider$_{16w}$, Poptrie$_{18}$, and D18R are in proportion to the CPU frequency; the result indicates that the processing of the CPU instructions is dominant to their performance. For IPv4, when the CPU frequency is 1.0 GHz, Spider$_{16w}$ reaches 158 Mlps, which is 1.8 times faster than DIR-24-8. DIR-24-8 is the second-fastest method at that frequency. In addition, when the CPU frequency is 2.1 GHz, Spider$_{16w}$ reaches 318 Mlps, which is 2.7 times faster than DIR-24-8. Note that the performance of DIR-24-8 falls below the others when the CPU frequency is higher than 1.5 GHz, although it outperforms Poptrie$_{18}$ and D18R at lower CPU frequencies; the result indicates that the processing of the CPU instructions is not dominant to the performance of DIR-24-8 when CPU frequency is high. For IPv6, when the CPU frequency is 1.0 GHz, Spider$_{16w}$ reaches 58 Mlps, which is 2.3 times faster than DIR-24-8. When the CPU frequency is 2.1 GHz, Spider$_{16w}$ reaches 117 Mlps, which is 2.3 times faster than DIR-24-8. The trend shown in the result is mostly the same as for IPv4, except that DIR-24-8 is also in proportion to the CPU frequency, and shows its performance advantage over Poptrie$_{18}$, even at higher CPU frequencies.

F. SCALABILITY IN A MULTICORE ENVIRONMENT

To confirm that the lookup rate of Spider scales along with the number of CPU cores as with the CPU frequency, we evaluated how the lookup rate changes depending on the number of cores. The scalability according to the number of CPU cores is worth evaluation because recent packet forwarding mechanisms are generally designed and implemented to scale up with the number of CPU cores [3], [22]. In this experiment, all CPU cores share a routing table as with the real-world packet processing software. Theoretically, the lookup rate would scale up to the bandwidth to the locations of the routing table. The location of the routing table varies from the CPU cache to the main memory depending on its memory footprint; however, in any case, the bandwidths of the locations are sufficient to accommodate all cores in current CPUs. From Table 1, the Poptrie$_{18}$ and D18R fit their routing table in the size of the L1 data CPU cache. Spider$_{16w}$'s routing table is larger than that of Poptrie$_{18}$ and D18R, although it fits its routing table in the L3 CPU cache at least. DIR-24-8 has a relatively large routing table compared with other methods, which would cause frequent reference to the main memory. Consequently, all methods would scale up with the number of CPU cores in current CPUs, although the rate would vary depending on the design of each method.

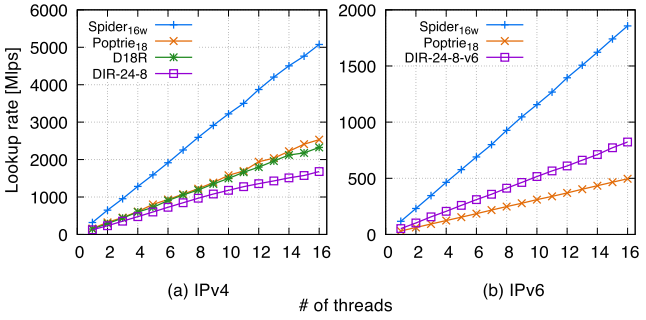

FIGURE 12. Lookup rates of Poptrie$_{18}$, D18R, DIR-24-8, and Spider with multiple CPU cores up to 16 CPU cores.

Figure 12 shows the lookup rates of Poptrie$_{18}$, D18R, DIR-24-8, and Spider$_{16w}$ with multiple CPU cores up to 16. Through the overall result, we can observe that all methods scale with the number of CPU cores. In this measurement, we disabled Intel's turbo-boost because it causes a biased acceleration of the CPU frequency, which leads to a fluctuation of the result. For IPv4, with a single CPU core, Spider$_{16w}$ achieves 318 Mlps, which overcomes the wire-rate of 200 Gbps. With 16 CPU cores, Spider$_{16w}$, Poptrie$_{18}$, D18R, and DIR-24-8 reach 5,074 Mlps, 2,532 Mlps, 2,325 Mlps, and 1,680 Mlps, respectively. The lookup rate of Spider$_{16w}$ with 16 CPU cores is equivalent to 34 ports of 100 Gbps interface, while that of Poptrie$_{18}$ is equivalent to 17 ports. For IPv6, with a single CPU core, Spider$_{16w}$ achieves 117 Mlps, which almost equals the capacity of 80 Gbps. With 16 CPU cores, Spider$_{16w}$, Poptrie$_{18}$, and DIR-24-8 reach 1,857 Mlps, 496 Mlps, and 823 Mlps, respectively. The lookup rate of Spider$_{16w}$ with 16 CPU cores for IPv6 is equivalent to 12 ports of 100 Gbps interface.

VI. CONCLUSION

In this paper, we have proposed Spider, which achieves an improvement of the LPM performance by parallelizing its lookup procedure in a single CPU core. We have achieved a fully parallelized LPM procedure by designing a data structure of a routing table optimized for SIMD instructions. The advantage of using SIMD instructions is that it does not require any additional hardware because the instructions are commonly included in recent CPUs. The key to achieving the fully parallelized procedure of LPM is to utilize the `gather` instruction, which enables a single CPU core to execute LPM in parallel. As a result, Spider demonstrated a dramatic improvement in LPM performance compared with the state-of-the-art software LPM methods. The evaluation shows that Spider$_{16w}$ improves the lookup rate by 1.8–3.2 times compared with the state-of-the-art methods. In addition, Spider$_{16w}$ achieves 5,074 Mlps with 16 CPU cores,

which is equivalent to 34 ports of 100 Gbps interface. This work opens up the possibility of applying software network applications for packet processing at the terabit-class rate.

ACKNOWLEDGMENT

The authors would like to thank Assoc. Prof. K. Fukuda with the National Institute of Informatics, Japan, for his valuable advice.

REFERENCES

[1] R. Guerzoni, "Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action, introductory white paper," in *Proc. SDN OpenFlow World Congr.*, vol. 1, 2012, pp. 5–7.
[2] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Commun. Mag.*, vol. 53, no. 2, pp. 90–97, Feb. 2015.
[3] Y. Ohara, Y. Yamagishi, S. Sakai, A. D. Banik, and S. Miyakawa, "Revealing the necessary conditions to achieve 80Gbps high-speed PC router," in *Proc. Asian Internet Eng. Conf. (AINTEC)*, 2015, pp. 25–31.
[4] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM Conf.*, 2013, pp. 15–26.
[5] S. Sahni and H. Lu, "Dynamic tree bitmap for IP lookup and update," in *Proc. 6th Int. Conf. Netw. (ICN)*, Apr. 2007, p. 79.
[6] H. Song, M. Kodialam, F. Hao, and T. V. Lakshman, "Scalable IP lookups using shape graphs," in *Proc. 17th IEEE Int. Conf. Netw. Protocols*, Oct. 2009, pp. 73–82.
[7] H. Le and V. K. Prasanna, "Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 1026–1039, Jul. 2012.
[8] K. Huang, G. Xie, Y. Li, and A. X. Liu, "Offset addressing approach to memory-efficient IP address lookup," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 306–310.
[9] G. Rétvçri, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: Towards entropy bounds and beyond," *ACM SIGCOMM Computer Commun. Rev.*, vol. 43, no. 4, pp. 111–122, 2013.
[10] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 57–70, Aug. 2015.
[11] M. Zec, L. Rizzo, and M. Mikuc, "DXR: Towards a billion routing lookups per second in software," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 5, pp. 29–36, Sep. 2012.
[12] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee IP lookup performance with FIB explosion," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 39–50.
[13] J. L. Hennessy and D. A. Patterson. (Feb. 2019). *A New Golden Age for Computer Architecture*. [Online]. Available: https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext
[14] Y. Ueno, R. Nakamura, Y. Kuga, and H. Esaki, "Spider: Parallelizing longest prefix matching with optimization for SIMD instructions," in *Proc. 6th IEEE Conf. Netw. Softwarization (NetSoft)*, Jun. 2020, pp. 267–271.
[15] F. L. Faucheur, L. Wu, B. Davie, S. Davari, P. Vaananen, R. Krishnan, P. Cheval, and J. Heinanen, *Multi-Protocol Label Switching (MPLS) Support of Differentiated Services*, document RFC3270, 2002.
[16] J. W. Evans and C. Filsfils, *Deploying IP and MPLS QoS for Multiservice Networks: Theory and Practice*. Amsterdam, The Netherlands: Elsevier, 2010.
[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000.
[18] L. Rizzo, "Netmap: A novel framework for fast packet I/O," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 101–112.
[19] L. Rizzo and G. Lettieri, "VALE, a switched Ethernet for virtual machines," in *Proc. 8th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2012, pp. 61–72.
[20] M. Honda, F. Huici, G. Lettieri, and L. Rizzo, "mSwitch: A highly-scalable, modular software switch," in *Proc. ACM SIGCOMM Symp. Softw. Defined Netw. Res. (SOSR)*, 2015, pp. 1–13.
[21] DPDK Project. (2020). *DPDK: Home*. [Online]. Available: https://www.dpdk.org
[22] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2015, pp. 5–16.
[23] J. Gil Herrera and J. F. Botero, "Resource allocation in NFV: A comprehensive survey," *IEEE Trans. Netw. Service Manage.*, vol. 13, no. 3, pp. 518–532, Sep. 2016.
[24] V.-G. Nguyen, A. Brunstrom, K.-J. Grinnemo, and J. Taheri, "SDN/NFV-based mobile packet core network architectures: A survey," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1567–1602, 3rd Quart., 2017.
[25] A. J. McAuley and P. Francis, "Fast routing table lookup using CAMs," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Mar. 1993, pp. 1382–1391.
[26] F. Zane, G. Narlikar, and A. Basu, "Coolcams: Power-efficient TCAMs for forwarding engines," in *Proc. IEEE 22nd Annu. Joint Conf. Comput. Commun. Societies (INFOCOM)*, vol. 1, Mar. 2003, pp. 42–52.
[27] K. Zhen, C. Hu, H. Lu, and B. Liu, "A TCAM-based distributed parallel IP lookup scheme and performance analysis," *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 863–875, Aug. 2006.
[28] W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup," in *Proc. IEEE INFOCOM 27th Conf. Comput. Commun.*, Apr. 2008, pp. 1786–1794.
[29] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani, "Scalable, memory efficient, high-speed IP lookup algorithms," *IEEE/ACM Trans. Netw.*, vol. 13, no. 4, pp. 802–812, Aug. 2005.
[30] M. Bando and H. J. Chao, "FlashTrie: hash-based prefix-compressed trie for IP route lookup beyond 100Gbps," in *Proc. IEEE INFOCOM*, Mar. 2010, pp. 1–9.
[31] H. Lim, W. Kim, B. Lee, and C. Yim, "High-speed IP address lookup using balanced multi-way trees," *Comput. Commun.*, vol. 29, no. 11, pp. 1927–1935, Jul. 2006.
[32] H. Fadishei, M. S. Zamani, and M. Sabaei, "A novel reconfigurable hardware architecture for IP address lookup," in *Proc. Symp. Archit. Netw. Commun. Syst. (ANCS)*, 2005, pp. 81–90.
[33] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Netw.*, vol. 15, no. 2, pp. 8–23, Mar. 2001.
[34] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small forwarding tables for fast routing lookups," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 3–14, Oct. 1997.
[35] V. Srinivasan and G. Varghese, "Fast address lookups using controlled prefix expansion," *ACM Trans. Comput. Syst.*, vol. 17, no. 1, pp. 1–40, Feb. 1999.
[36] P. Crescenzi, L. Dardini, and R. Grossi, "IP address lookup made fast and simple," in *Proc. Eur. Symp. Algorithms*, in Lecture Notes in Computer Science, vol. 1643, 1999, pp. 65–76.
[37] S. Nilsson and G. Karlsson, "IP-address lookup using LC-tries," *IEEE J. Sel. Areas Commun.*, vol. 17, no. 6, pp. 1083–1092, Jun. 1999.
[38] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: Hardware/software IP lookups with incremental updates," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, Apr. 2004.
[39] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *Proc. IEEE INFOCOM Conf. Comput. Commun. 17th Annu. Joint Conf. IEEE Comput. Commun. Soc. Gateway 21st Century*, vol. 3, Mar. 1998, pp. 1240–1247.
[40] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," *IEEE/ACM Trans. Netw.*, vol. 14, no. 2, pp. 397–409, Apr. 2006.
[41] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, "IPv6 lookups using distributed and load balanced Bloom filters for 100Gbps core router line cards," in *Proc. IEEE INFOCOM 28th Conf. Comput. Commun.*, Apr. 2009, pp. 2518–2526.
[42] H. Lim, K. Lim, N. Lee, and K.-H. Park, "On adding Bloom filters to longest prefix matching algorithms," *IEEE Trans. Comput.*, vol. 63, no. 2, pp. 411–423, Feb. 2014.
[43] P. Kennedy. (Jul. 2017). *Intel Xeon Scalable Processor Family Microarchitecture Overview*. [Online]. Available: https://www.servethehome.com/intel-xeon-scalable-processor-family-microarchitecture-overview/
[44] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proc. Conf. ACM Special Interest Group Data Commun.*, Aug. 2018, pp. 327–341.

[45] Y. Li, D. Zhang, A. X. Liu, and J. Zheng, "GAMT: A fast and scalable IP lookup engine for GPU-based software routers," in *Proc. Archit. Netw. Commun. Syst.*, Oct. 2013, pp. 1–12.
[46] T. Yang, G. Xie, A. X. Liu, Q. Fu, Y. Li, X. Li, and L. Mathy, "Constant IP lookup with FIB explosion," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1821–1836, Aug. 2018.
[47] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 195–206, 2011.
[48] J. Zhao, X. Zhang, X. Wang, Y. Deng, and X. Fu, "Exploiting graphics processors for high-performance IP lookup in software routers," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 301–305.
[49] S. Zhou and V. K. Prasanna, "Scalable GPU-accelerated IPv6 lookup using hierarchical perfect hashing," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2015, pp. 1–6.
[50] A. Ozsoy, "An efficient parallelization of longest prefix match and application on data compression," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 3, pp. 276–289, Aug. 2016.
[51] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as packet processing accelerator," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 83–96.
[52] Intel Corporation. (2020). *Intel Architecture Instruction Set Extensions and Future Features Programming Reference*. [Online]. Available: https://software.intel.com/content/dam/develop/public/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf
[53] S. Muchnick, *Advanced Compiler Design and Implementation*. San Mateo, CA, USA: Morgan Kaufmann, 1997.
[54] Cisco Systems. (2009). *BGP Commands on Cisco IOS XR Software*. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/routers/crs/software/crs_r3-9/roting/command/reference/rr39crs1book_chapter1.html
[55] P. E. McKenney and J. Walpole. (2007). *What is RCU, fundamentally?* [Online]. Available: https://lwn.net/Articles/262464/
[56] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill, "Constructing optimal IP routing tables," in *Proc. IEEE INFOCOM Conf. Comput. Commun. 18th Annu. Joint Conf. IEEE Comput. Commun. Societies Future Now*, Mar. 1999, pp. 88–97.
[57] APNIC. (2020). *BGP in 2019*. [Online]. Available: https://blog.apnic.net/2020/01/14/bgp-in-2019-the-bgp-table/
[58] M. Zec. (2019). *DXR: Direct/Range Routing Lookups*. [Online]. Available: http://www.nxlab.fer.hr/dxr/
[59] H. Asai. (2019). *Pixos/Poptrie: An Implementation of Poptrie IP Routing Table Lookup Algorithm*. [Online]. Available: https://github.com/pixos/poptrie
[60] WIDE Project. (2019). *Packet Traces From WIDE Backbone*. [Online]. Available: http://mawi.wide.ad.jp/mawi
[61] Intel Corporation. (2020). *Intel Xeon Processor Scalable Family Specification Update*. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf
[62] Intel DPDK Validation Team. (2018). *DPDK Intel NIC Performance Report Release 18.02*. [Online]. Available: https://fast.dpdk.org/doc/perf/DPDK_18_02_Intel_NIC_performance_report.pdf

YUKITO UENO received the master's degree in media and governance from Keio University, Tokyo, Japan, in 2016. He is currently pursuing the Ph.D. degree with the Graduate School of Information Science and Technology, The University of Tokyo.

He is also an Engineer with the Department of Innovation Center, NTT Communications Corporation. His research interests include network architecture and software packet processing.

RYO NAKAMURA received the Ph.D. degree in information science and technology from The University of Tokyo, Tokyo, Japan, in 2017.

He is currently a Research Associate with the Information Technology Center, The University of Tokyo. His research interests include networking aspect in operating systems, network virtualization, and network operation.

YOHEI KUGA received the Ph.D. degree in media and governance from Keio University, Tokyo, Japan, in 2015.

He is currently a Project Lecturer with the Information Technology Center, The University of Tokyo. His current research interests include systems aspects of networking hardware and high-performance computing and networking.

HIROSHI ESAKI (Member, IEEE) received the Ph.D. degree in electronic engineering from The University of Tokyo, Tokyo, Japan, in 1998.

In 1987, he joined the Research and Development Center, Toshiba Corporation. From 1990 to 1991, he was with the Applied Research Laboratory, Bellcore Inc., as a Residential Researcher. From 1994 to 1996, he was with the Center for Telecommunication Research, Columbia University, New York, NY, USA. In 1998, he was a Professor with The University of Tokyo.

Dr. Esaki is an Executive Director of the IPv6 Promotion Council and the Vice President of JPNIC. He is also an IPv6 Forum Fellow and a Board Member of the WIDE Project and the Board of Trustees of the Internet Society.