Spider: Parallelizing Longest Prefix Matching with Optimization for SIMD Instructions

Yukito Ueno
The University of Tokyo / NTT Communications
eden@g.ecc.u-tokyo.ac.jp

Ryo Nakamura
The University of Tokyo
upa@nc.u-tokyo.ac.jp

Yohei Kuga
The University of Tokyo
sora@nc.u-tokyo.ac.jp

Hiroshi Esaki
The University of Tokyo
hiroshi@wide.ad.jp

***Abstract*—Longest prefix matching (LPM) is a fundamental process in IP routing used not only in traditional hardware routers but also in modern software middleboxes such as the applications of Network Function Virtualization. However, the performance of recent LPM methods in software routers is insufficient for high-speed packet processing such as two or more 100 Gbps throughput. To improve the performance of LPM, we propose Spider, a new LPM method that achieves a fully parallelized LPM procedure using single instruction, multiple data (SIMD) instructions in a CPU. The evaluation shows that the proposed method has 1.8–1.9 times faster LPM performance compared with the state-of-the-art methods in this study area. We describe the Spider's lookup procedure fully parallelized by SIMD instructions and the design of the routing table effectively processed by the procedure. We also report the following three evaluations: (1) how parallelism by the SIMD instructions contributes to performance; (2) the scalability of Spider with the number of CPU cores; and (3) the performance comparison with the previous methods in terms of randomly generated and real-trace traffic patterns.**

***Index Terms*—IP Routing, LPM, SIMD**

I. Introduction

Longest prefix matching (LPM) is a fundamental function of IP routing in both software middleboxes and hardware routers. Modern routers need to process LPM extremely fast because of the significant increase of network interface speed. Many hardware routers used in current commercial networks process the LPM by ternary content addressable memory to achieve high LPM performance. On the other hand, the demand for high-speed packet processing in software middleboxes is increasing because software-based approaches enable developers to make further advanced packet processing software. For example, the network function virtualization [1], [2], software routers for backbone networks [3], and the software-defined wide-area network [4] mostly require software-based LPM implementation.

However, the performance of recent LPM methods in software routers is insufficient for high-speed packet processing such as two or more 100 Gbps throughput. A recent approach to improve the performance of LPM is to leverage CPU cache to minimize the access speed to the routing table [5]–[7]. To keep the data cache hot, these methods fit their routing tables in the CPU caches by the optimization of the data structure. Another approach is to leverage parallelism with accelerators such as GPUs to improve the throughput of LPM [8]–[10]. These methods exploit the parallelism of GPUs powered by the abundant memory bandwidth and conceal the data transfer latency between the CPU and the GPUs by overlapping data transfer and processing.

To improve the performance of LPM while exploiting the low latency of the CPU, we propose a fully parallelized LPM procedure using single instruction, multiple data (SIMD) instructions. We have applied techniques to parallelize LPM, which has been achieved in GPUs in previous work, to the processing in the CPU. Our method provides an 8-way parallel LPM procedure by the CPU using the SIMD instructions. In addition, we have optimized the procedure by combining the two iterations into a single loop to conceal the latency of memory access, which results in 16-way lookups per iteration. The proposed method achieves a 1.8–1.9 times faster lookup rate of LPM processing, compared with the state-of-the-art methods for the CPU, with random traffic patterns and routes of the border gateway protocol (BGP) in two real ISPs. The contributions of this paper are as follows.

- We have applied techniques to parallelize LPM in GPUs to the processing in the CPU using SIMD instructions.
- We have achieved an 80% or higher lookup rate improvement compared with the state-of-the-art methods for CPU.

II. Related work

Approaches for fast LPM in software can be categorized as leveraging CPU cache and leveraging accelerators. This section describes their characteristics and resulting performances.

Leveraging CPU cache for fast LPM in software is a recent major approach [5]–[7]. In this approach, the methods exploit the low latency of access speed to the CPU cache by keeping the entire routing table on the CPU cache using dedicated data structures. The characteristics of each data structure differ greatly depending on the methods. For example, Poptrie [5] uses a variant of a multiway trie that can omit unnecessary child nodes in the memory area. The key idea of Poptrie is to use the `popcnt` instruction to count the number of two types of child nodes from a single bit vector, to omit either type of child nodes from the memory area without performance drawback. As a result, Poptrie has achieved 240.52 million lookups per second (Mlps) with a random traffic pattern and a BGP full route as of 2015. DXR [6] also takes a similar

approach to Poptrie, which has achieved 179.92 Mlps in the evaluation of Poptrie's paper. The rate of lookup indicates the potential capacity to process the same rate of packet processing, whose rate is described in the form of packets per second (pps) or bits per second (bps). Thus, the performance of Poptrie's 240.52 Mlps indicates the potential capacity to process 240.52 Mpps, which is over the theoretical upper limit of the packet processing rate of 100 Gbps Ethernet.

Another approach is to exploit accelerators such as GPUs to increase lookup throughput by executing LPM in parallel [8]–[10]. To utilize the parallelism capacity of GPUs, the methods tend to use a table, which is called a state-jump table, converted from common data structures representing a routing table such as multiway trie. In the state-jump table, each element of the original trie node is encoded into an integer according to several rules. The reason why the methods use the state-jump table are (1) to avoid the performance drawback caused by conditional branching by encoding the condition into the location of the element, and (2) to arrange the data in a form that GPUs can look up efficiently. As a result of the characteristics of GPUs and the corresponding optimizations, GAMT [8] has achieved over 1,000 Mlps when 2^{24} destination IP addresses are prepared in advance of the execution, while the lookup rate declines to 50 Mlps when the prepared number of destination IP addresses is less than 2^{12}.

On the other hand, both approaches have disadvantages. For the methods that exploit low latency of the CPU cache, more improvement to the approach would be brought by the increase of CPU clock rate although it has stagneted [11], because the part of accessing data of these methods is already optimized. For the methods that exploit accelerators, the dedicated hardware increases the cost for addressing power consumption [12], and the latency to complete each lookup.

III. Proposed method

For faster LPM in software while overcoming the drawbacks, we propose a novel LPM method exploiting SIMD instructions, called `Spider`. The main idea of Spider is to apply techniques to parallelize LPM, which has been achieved in GPUs in previous work, to the processing in the CPU. We adopt a state-jump table [8] as the data structure to represent a routing table, and optimize it to be effectively processed with Spider's LPM procedure that consists of SIMD instructions. The gather instruction is the key part to achieve our method, which loads multiple data located in split areas by a single instruction. The recently implemented gather instruction can be used to implement the table lookup procedure, which is the last piece to fully parallelize LPM by SIMD instructions. By using these techniques, we demonstrate that the parallelization of LPM by SIMD instructions produces a higher lookup rate.

In real packet processing softwares, maximizing the performance of parallelized LPM needs to process multiple packets at once by waiting arrival of these packets. Consequently, the waiting time increases the latency of each packet. However, fortunately, modern packet processing software uses the technique called packet batching [3], [10], [13], [14], which means processing multiple packets per function. These softwares can be easily extended to process the packets in parallel without any drawback because packet batching and parallelization are common in handling multiple packets at once. Therefore, parallelizing LPM is reasonable in the packet processing softwares with packet batching.

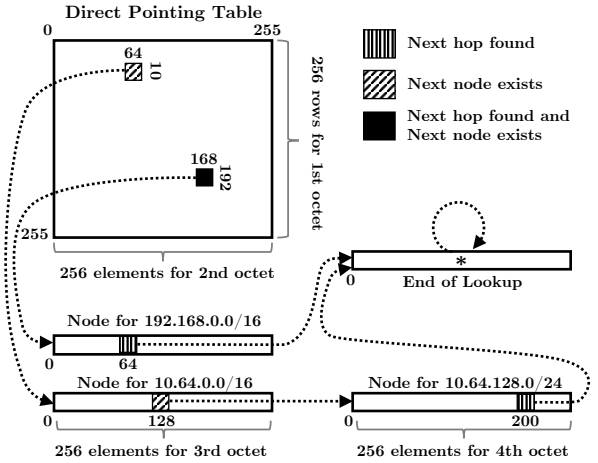

Fig. 1. Design of Spider's data structure: an example of a routing table in which 192.168.0.0/16, 192.168.64.0/24, and 10.64.128.192/28 are installed, and 192.168.64.1 and 10.64.128.200 are processed.

A. Routing table design of Spider

We adopt the state-jump table as the data structure of Spider, which was originally proposed for LPM on GPUs. The SIMD instructions require that the data structure must be packed in one area and byte-aligned on the memory. Because of the requirements, the SIMD instructions cannot handle the data structures previously proposed for processing LPM in CPU. To satisfy the requirement, we adopt the fixed-length stride of the state-jump table while the original method adopts the variable-length stride.

The state-jump table of Spider, which is shown in Figure 1, is constructed by arranging each node in the original multiway trie into a two-dimensional array. A multiway trie is widely used as a data structure for routing tables. In both the multiway trie and the state-jump table of Spider, each node contains 256 elements and the elements correspond to a possible value of an 8-bit part of the IP address. For the state-jump table, each element contains two types of 16-bit length information: next hop (NH) and next node (NN). NH represents the index number of next hop information managed separately and NN represents the location of the child node, which is the number of the row. The detailed procedure of conversion is as follows: (1) convert each node of the multiway trie into a single array of the 256 elements; (2) fill the elements by converting pointers to children in the multiway trie to the number of the row where the target node will be located; and (3) arrange the rows into a two-dimensional array together with the End-of-Lookup and the Direct Pointing Table.

In the situation shown in Figure 1, there are three routes: 192.168.0.0/16, 192.168.64.0/24, and 10.64.128.192/28. When the destination IP address is 192.168.64.1, the lookup procedure is as follows. (1) The lookup procedure fetches the element corresponding to the 1st and 2nd octets of the IP address from the Direct Pointing Table, which is a well-known optimization described in Section III-C. In this example, because the element contains an NN for the node representing 192.168.0.0/16, the lookup procedure will use the node to search longer routes. In addition, because the element also contains an NH for 192.168.0.0/16, the lookup procedure saves the NH as the result. (2) The lookup procedure fetches the element corresponding to the 3rd octet of the IP address from the node representing 192.168.0.0/16. At this time, because the 64th element of the node for 192.168.0.0/16 does not contain the NN, the lookup procedure detects the completion of the lookup. On the other hand, because the 64th element contains an NH for 192.168.64.0/24, the lookup procedure overwrites the result with the NH. The NH will be the longest matched result because the following procedure will not find a new result in this case.

Because of the mechanism of SIMD, Spider need to continue executing the same instructions until lookup of all destination IP addresses complete. To address the requirement, Spider uses End-of-Lookup. A lookup for an IP address that ended earlier than others continues to go around End-of-Lookup because all elements in End-of-Lookup contain zero as the value of NN and NH. Therefore, in this example, if lookups for any other IP addresses have not ended, the lookup procedure will fetch the 1st element of the End-of-Lookup node according to the 4th octet of the IP address.

B. Parallelization by SIMD instructions

Spider looks up multiple destination IP addresses simultaneously because the entire algorithm consists of SIMD instructions. In the lookup phase, destination IP addresses are divided into 8-bit parts, and Spider traces the elements on the state-jump table according to each 8-bit part of the destination IP addresses in parallel. We assume AVX2 by Intel as the primary instruction set to implement Spider. However, other instruction sets can also be used because the design of Spider is independent of the specific instruction set. For the implementation by AVX2, Spider looks up eight destination IP addresses in parallel, because AVX2 can address up to 256-bit data.

Algorithm 1 describes the lookup procedure of Spider at the instruction level. The subscripts of each operation represent the data type of SIMD instructions. The basic procedure is as follows: (1) extract the first 16-bit part for direct pointing or next 8-bit part for normal iteration from the destination IP addresses by `shuffle` instruction; (2) `add` the extracted parts to the value of $NN \times 2^{stride}$, where the stride is fixed to 8 in Spider, to calculate the indexes of the next elements; (3) fetch the next elements based on the indexes by `gather` instruction; and (4) extract NH and NN from the elements by `shuffle` instruction. The lookup procedure ends when all NNs indicate zero.

Algorithm 1 Lookup procedure of Spider

Input: $DstArray$
Output: $ResArray$
1: $load_{256}(dst, DstArray)$;
2: /* Direct pointing for first 16-bit parts of IPs */
3: $idx = shuffle_8(dst, maskd16)$;
4: $idx = add_{32}(idx, 256)$; // row[1] + idx
5: $val = gather_{32}(fib, idx)$;
6: $nh = shuffle_8(val, masknh)$;
7: $res = nh$;
8: $nn = shuffle_8(val, masknn)$;
9: **while** *not all NNs are 0* **do**
10: /* Iterative lookup for each 8-bit parts of IPs */
11: $idx = shuffle_8(dst, maskd8)$;
12: $idx = add_{32}(idx, nn)$; // row[NN] + idx
13: $val = gather_{32}(fib, idx)$;
14: $nh = shuffle_8(val, masknh)$;
15: $maskbl = cmpeq_{32}(nh, 0)$;
16: $res = blend_{32}(maskbl, res, nh)$;
17: $nn = shuffle_8(val, masknn)$;
18: **end while**
19: $store_{256}(ResArray, res)$;
20: $return$;

C. Optimization

In addition to the basic design of the data structure and procedure, we applied the following well-known optimization techniques to Spider, to further enhance the performance: (1) route aggregation; (2) direct pointing; and (3) loop fusion. For a preliminary optimization, we applied route aggregation [15] to the multiway trie to reduce the size of the resulting state-jump table. Route aggregation is a common technique to combine a plurality of routes that have the same next hop into one route. Route aggregation reduces the size of a routing table, and is applicable to other LPM methods. In addition, we used a technique to reduce the number of traversals of the routing table, called direct pointing [5]. 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. In Spider, the array is prepared for 16-bit length and is arranged as a part of the state-jump table from the 1st row to the 256th row. In the lookup algorithm of Spider, we combined two iterations of a lookup into a single loop to conceal the latency of memory access of load, store, and gather instructions. This well-known optimization is called loop fusion [16]. As a result, the maximum number of destination IP addresses processed in a single iteration becomes 16-way in Spider.

IV. Evaluation

We compare the performance of Spider with other methods, i.e., DXR and Poptrie. Parallelizing LPM by exploiting SIMD instructions would provide a higher lookup rate than other methods in spite of two major drawbacks: the larger memory footprint and the reduction of the CPU clock rate. As shown in Table I, the routing table of Spider has a larger memory

TABLE I
MEMORY FOOTPRINT OF ROUTING TABLES.

	ISP-A	ISP-B
Method	Memory size [MiB]	Memory size [MiB]
Spider	9.32	11.61
Poptrie$_{18}$	1.25	1.35
Poptrie$_{16}$	1.63	2.03
D18R	1.29	1.40
D16R	0.50	0.61

footprint than DXR and Poptrie because Spider uses a fixed-length stride in the state-jump table. By contrast, other methods tend to minimize the memory footprint by the optimization of the data structure including variable-length stride. As the result, it is expected that the entire routing table of Spider would not fit in the L1 cache. In addition, according to the specification [17], Intel's CPU used in this evaluation reduces its clock rate by 19% when processing AVX2 instructions to reduce the power consumption.

We used the implementation of DXR and Poptrie published as a module of Click modular router [18], [19] and a standalone library [20], respectively, with modification to measure the performance. In this evaluation, we applied route aggregation and direct pointing for all methods. Both DXR and Poptrie vary regarding the length of direct pointing, whose length is represented as the name, such as Poptrie$_{\mathbf{18}}$ and D**18**R, in a form aligned with previous work [5], [6]. The equipment used for experiments consists of an Intel(R) Xeon(R) Gold 6130 (3.7 GHz, 22 MiB cache) and 48 GB DDR4-2666 memory. We conducted measurements of all evaluations 10 times on the Ubuntu 18.04.3 LTS server (x86-64) on the equipment. The sizes of L1, L2, and L3 cache 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 called the BGP full route, which is suitable for this experiment because IP routing based on the BGP full route is the highest load situation that current routers face in real environments. As the current BGP routes of the Internet, we used two BGP full routes of real ISPs: ISP-A, and ISP-B. Both BGP full routes were captured on December 10, 2019.

We consider the `random` and `real` traffic patterns for the evaluation in this paper. For the random traffic pattern, we measured the time of looking up 2^{32} random destination IP addresses with just-in-time generation of the pattern aligned with Poptrie's paper. We used the linear congruential method to generate the random traffic pattern. For the real traffic pattern, we measured the time of looking up destination IP addresses from real Internet traffic captured on April 10, 2019, on the samplepoint-F of the WIDE backbone, which is published as the MAWI dataset [21]. For this pattern, the 2^{28} IP addresses are arranged in a 1-Gbyte array in advance as the maximum number that can be prepared without affecting the lookup performance.

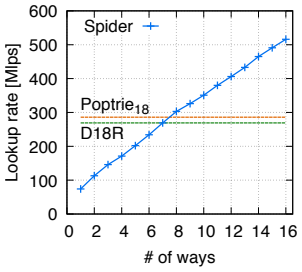

Fig. 2. Performance scale with the parallelism by the SIMD instructions in Spider.

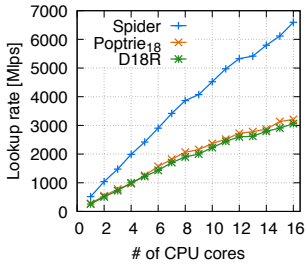

Fig. 3. Performance scale with the number of CPU cores in Spider.

B. Effect of parallelization in Spider

We evaluated how parallelism by the SIMD instructions contributes to performance, given that parallelizing LPM is our main contribution. The measurement was conducted with the random traffic pattern and the BGP full route of ISP-A.

According to the result shown in Figure 2, we confirmed the throughput of LPM scales along with the parallelism provided by the SIMD instructins. When processing the same number of IP addresses, the number of iterations decreases according to the degree of parallelism in Spider. The linear scaling in the result indicates that the reduction of the number of lookup iterations contributes to the performance. When the degree of parallelism is 16-way, Spider reaches 516 Mlps of lookup rate, which is 6.9 times faster than the case when the parallelism is 1-way, 1.8 times faster than Poptrie$_{18}$, and 1.9 times faster than D18R. The lookup rates of Spider and other methods become almost equal when Spider is between 7-way and 8-way.

When the parallelism of Spider is less than 7-way, the lookup rate of Spider falls below the others. A major factor is the cache hit rate in the CPU, which decreases along with the size of memory footprint. For Spider, the memory footprint is larger than 1 MiB of L1 data cache and other methods as shown in Table I. The other major factor of Spider's performance drawback at lower parallelism is the reduction of clock rate. In summary, for Spider, the reduction of the CPU cache hit ratio due to the size of the memory footprint and the reduction of the clock rate lead to lower performance at less than 7-way.

C. Scalability in a multicore environment

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], [14]. Thus, we evaluated how the lookup rate changes depending on the number of cores. In this experiment, all CPU cores share the routing table. From Table I, all methods fit the memory footprint of their routing table at least in the L3 cache. Therefore, the performance of all methods should scale with the number of cores until the bandwidth of the L3 cache runs out.

Figure 3 shows the result of the experiment. The lookup rates of all of the methods including Poptrie$_{18}$, D18R, and

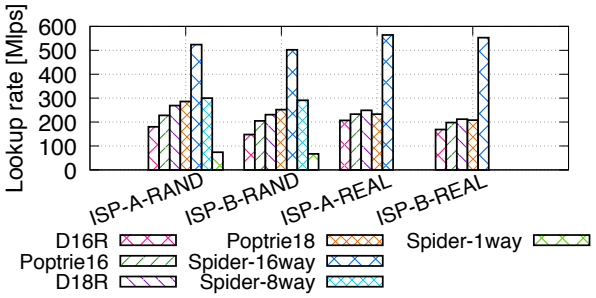

Fig. 4. Lookup rate comparison with other methods.

Spider scale up to the equipment limitation of 16 cores. Spider achieves 6,595 Mlps, Poptrie$_{18}$ achieves 3,205 Mlps, and D18R achieves 3,061 Mlps at 16 cores. The ratio of lookup rate differences between Spider and other methods remains almost unchanged as the number of cores increases. The lookup rate of Spider scales up to 12.8 times faster from 1-core to 16-core, and the lookup rate is sufficient for the throughput of two or more 100 Gbps link speed.

D. Comparison of lookup rate with other methods

To confirm the performance advantage of Spider, we compare the lookup rate of Spider with Poptrie$_{18}$, Poptrie$_{16}$, D18R, and D16R with the 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 instructions, while we measured only the 16-way version of Spider for the real-trace lookup for implementation restriction caused by the measurement condition.

Figure 4 shows the result of the random and real-trace traffic patterns. Through all experiments, Spider outperforms other methods for both the ISP-A and ISP-B BGP full routes. The results show that the parallelization by SIMD instructions leads to the performance improvement, which can compensate for the longer time required to load data in the CPU due to the larger memory footprint and the reduction of the clock rate. In comparison with Poptrie$_{18}$, which is the next highest rate of Spider, Spider is 1.8 times faster in random lookup with the BGP full route of ISP-A and 1.9 times faster with ISP-B. In terms of real-trace lookup, Spider is 2.42 times faster with the BGP full routes of ISP-A and 2.65 times faster with ISP-B.

V. Conclusion

In this paper, we have proposed Spider, which achieves the fully parallelized procedure of LPM by SIMD instructions to improve the performance of LPM in software. We have applied the techniques to parallelize LPM in GPUs to the processing in CPU. The key to achieving the fully parallelized procedure of LPM is to utilize the gather instruction, which enables the CPU to execute table lookup in a fully parallelized manner by using SIMD instructions. As a result, Spider demonstrated a dramatic improvement in LPM performance compared with the previous methods. The evaluation shows that the lookup rate improved by 80% or more in random lookup with the BGP full routes of real ISPs. The Spider's performance improvement of LPM takes various software-based network applications to the next step in which two or more 100 Gbps throughput are required as the mainstream of communication speed.

References

[1] N. F. V. ETSI, "Introductory white paper." Technical Report, SDN and OpenFlow World Congress, 2012.

[2] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 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," *Proceedings of the Asian Internet Engineering Conference on - AINTEC '15*, pp. 25–31, 2015.

[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 *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 15–26.

[5] H. Asai and Y. Ohara, "Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup," *2015 ACM SIGCOMM Conference on Special Interest Group on Data Communication*, pp. 57–70, 2015.

[6] 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, p. 29, 2012.

[7] M. Bando and H. J. Chao, "FlashTrie: Hash-based prefix-compressed trie for ip route lookup beyond 100Gbps," *Proceedings - IEEE INFOCOM*, 2010.

[8] Y. Li, D. Zhang, A. X. Liu, and J. Zheng, "GAMT: A fast and scalable IP lookup engine for GPU-based software routers," *ANCS 2013 - Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 1–12, 2013.

[9] J. Zhao, X. Zhang, X. Wang, Y. Deng, and X. Fu, "Exploiting graphics processors for high-performance IP lookup in software routers," *Proceedings - IEEE INFOCOM*, pp. 301–305, 2011.

[10] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 195–206, 2011.

[11] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," Feb 2019. [Online]. Available: https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext

[12] S. Collange, D. Defour, and A. Tisserand, "Power consumption of gpus from a software perspective," in *International Conference on Computational Science*. Springer, 2009, pp. 914–923.

[13] M. Honda, F. Huici, G. Lettieri, and L. Rizzo, "mswitch: A highly-scalable, modular software switch," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:13.

[14] T. Barbette, C. Soldani, and L. Mathy, "Fast userspace packet processing," *ANCS 2015 - 11th 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 5–16, 2015.

[15] R. Draves, C. King, S. Venkatachary, and B. D. Zill, "Constructing optimal ip routing tables," in *Infocom*, vol. 99, 1999, pp. 88–97.

[16] S. Muchnick *et al.*, *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[17] "Intel® xeon® processor scalable family specification update," Intel Corporation, 2019. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[19] M. Zec, "Dxr: Direct / range routing lookups." [Online]. Available: http://www.nxlab.fer.hr/dxr/

[20] "pixos/poptrie: An implementation of poptrie ip routing table lookup algorithm." [Online]. Available: https://github.com/pixos/poptrie

[21] "Packet traces from wide backbone," WIDE Project, 2019. [Online]. Available: http://mawi. wide. ad. jp/mawi