Speeding Up IP Lookup Procedure in Software Routers by Means of Parallelization

Mihailo Vesović, *Graduate Student Member, IEEE*, Aleksandra Smiljanić, *Member, IEEE*, and Milo Tomašević

***Abstract* — Performance of software routers is limited by the speed of the operating system network protocol stack. A faster network protocol stack can be implemented in user space by utilizing parallelization and different optimization techniques. In this work we demonstrate an efficient implementation of the IP lookup algorithm in user space with multibit trie structures. Afterwards, we demonstrate the improvements achieved through parallelization. We evaluate pthreads and OpenMP parallelization methods and compare their performance.**

***Keywords* — 10 GbE, high speed packet I/O, IP lookup, parallelization, software routers.**

I. Introduction

Routers are fundamental devices in modern networks. Vendors aim to provide the devices which are easy to configure and install, hiding their operations from users. However, there is a need for more flexible solutions, where users will be able to modify the routing process according to their own needs. For this reason, research and industry are showing interest for the routers that are open and implemented in software.

A software router is a server with built-in network interface cards (NIC) and installed routing software. Software routers have multiple advantages over their hardware counterparts – simplicity of adding or modifying routing functionalities, low price and independence from hardware manufacturers. The main disadvantage of software routers is the performance [1].

Packet rate is a measure used to characterize speeds of software routers. It represents the number of packets routed per second. Maximal achievable routing performance of the Linux protocol stack is limited to 4 Mpps (Mega packets per second) [2]. In order to fully utilize 10 GbE cards, software routers must forward packets at rates of at least 14.88 Mpps per port. This is the worst case scenario, in which only 64B Ethernet frames are sent on the link.

There are various reasons why the Linux protocol stack is not capable of achieving higher routing speeds. The data path of the network protocol stack consists of many system calls [2]. Each system call represents an overhead of hundreds of nanoseconds, which is significant as system calls are invoked for each packet. Furthermore, packets are copied multiple times in the processing path, which is a time-consuming operation. Additionally, structures used to store packets contain many redundant fields in order to support various protocols, many of which are legacy protocols. As a result, the total number of instructions per packet is high, as well as the number of conditional instructions.

Speed of the existing Linux kernel is difficult to improve, so researchers implemented ways to bypass kernel and implement routing functionalities in user space. With fast I/O frameworks, such as netmap [2], DPDK [3] or psio [4], raw packets may be directly transferred from NIC to user-space. Implementing network protocol stack in user space brings multiple advantages. First, it is not necessary to have background in kernel development. Secondly, this method fits well with the virtualization of functionalities that offer flexible utilization of computing resources. Finally, better control of the packet forwarding can be achieved, and it is easier to parallelize routing applications.

In recent years, frequency scaling has reached its limit, and industry has moved to the processors with multiple cores [5]. In order to speed up routing process, it is not sufficient to optimize data processing path on a single core, but multiple processor cores must also be used. Thanks to the fact that NICs have multiple packet queues, it is possible to break data path processing across different independent threads, and to execute these threads in parallel.

The first part of this paper is related to the implementation of the IP lookup procedure in user space using the netmap framework. The second part of this paper is related to the parallelization of the IP lookup application. We have parallelized the lookup procedure in two different ways: using POSIX threads (pthreads) and using OpenMP API. At the end, we have compared the results. Parallelization has been designed for general-purpose processors.

Paper received May 25, 2017; accepted June 16, 2017. Date of publication July 31, 2017. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Grozdan Petrović.

This paper is a revised and expanded version of the paper presented at the 24th Telecommunications Forum TELFOR 2016 [9].

This work was supported by the Serbian Ministry of Science and Education (project TR-32022), and by companies Telekom Srbija and Informatika.

Mihailo Vesović is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11120 Belgrade, Serbia (e-mail: mikives@etf.rs).

Aleksandra Smiljanić is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11120 Belgrade, Serbia (e-mail: aleksandra@etf.rs).

Milo Tomašević is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11120 Belgrade, Serbia (e-mail: mvt@etf.rs).

II. Netmap Framework

Network cards comprise multiple ports, each with its own FIFO buffers for transmission and reception of packets. In the receive path, packets are forwarded from the input FIFO to one or more input queues. In the transmit path, packets are forwarded from one or more output queues to the output FIFO. Queues are organized in the form of circular buffers, i.e. NIC rings. Rings hold packet descriptors, which carry information about packet locations in memory, packet lengths and flags. The number of rings per port may vary, but there are usually as many rings as the processing cores.

Netmap is a framework for high speed packet I/O [2], which is based on replicas of NIC rings (Fig. 1). Netmap rings are circular buffers intended to be accessed by user space applications. Application accesses available slots, which are located between *head* and *tail* pointers. User space application reads packets from the Rx rings, and writes packets to the Tx rings.

Netmap framework performs synchronization between hardware NIC rings and software netmap rings. Synchronization is the procedure of slots' exchange between NIC rings and application rings. Netmap framework guarantees that user space application and NIC always use mutually exclusive sets of slots.

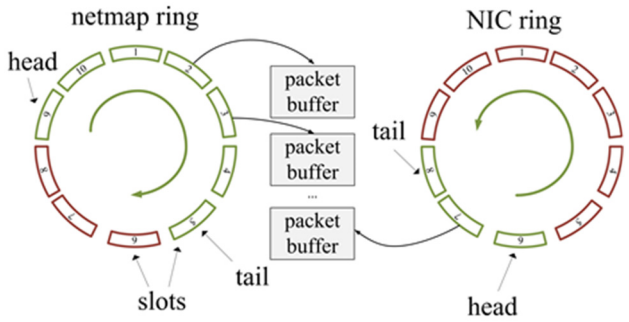

Fig. 1. Netmap and NIC rings. Each ring consists of slots that store packet descriptors.

Advantages brought by netmap allow faster routing speeds. Processing of the packets is performed in batches. System calls are initiated for each batch, which lowers the total system calls' overhead. Copying of packets is avoided, as it is time-consuming. For packet storage, simple pre-allocated structures are used. During the routing process, there will be no allocations.

III. Receive Side Scaling Mechanism

Receive-side scaling (RSS) is a mechanism which determines the way in which the received packets are distributed across different Rx rings according to contents of their headers [7]. The number of Rx rings varies from NIC to NIC. RSS is applicable only to IP packets. If the packets are non-IP, they will be sent to the first available Rx ring.

RSS mechanism is illustrated in Fig. 2, as implemented on the Intel X540 controller [7]. Hash value is calculated from source and destination IP addresses. If TCP/UDP headers are present, source and destination port numbers will also be included in the hash calculation. Calculated hash value is used as the input of the redirection table. Table is populated with the identificators of Rx rings to which the packets should be forwarded. It should be noted that packets cannot be distributed to input buffers in a round-robin fashion, as it would cause packet reordering.

Existence of multiple input buffers per port allows parallel packet processing. Usually, the number of Rx rings equals the number of cores in system. Application will spawn the number of threads equal to the number of cores multiplied by the number of ports. Each thread is in charge for one receive/transmit pair of queues at each port. All threads can independently process received packets on separate cores. The routing table is the only shared data, and its access must be protected.

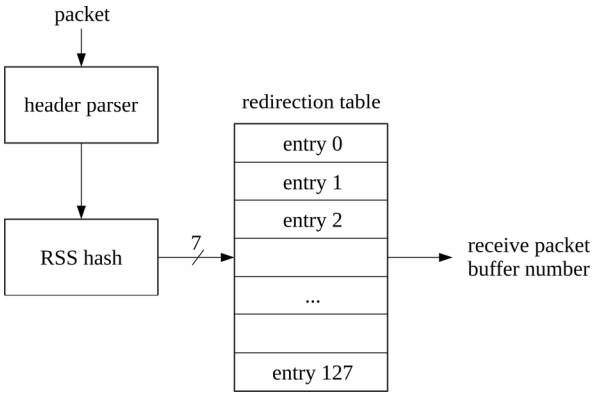

Fig. 2. Illustration of RSS mechanism [7].

IV. Parallelization Techniques

Parallelization may be achieved through the use of POSIX threads (pthreads). Pthreads have unique API on multiple operating systems, which guarantees interoperability [6]. It is a parallel programing at low level meaning that a programmer directly manipulates threads.

It is also possible to achieve parallelization at higher level, by using OpenMP API. OpenMP offers various benefits to programmers – parallelization simplicity, easy to read code, minimized possibility to cause race conditions, etc. However, finer control of the threads is lost, and debugging becomes complex.

In order to further improve performance, it is possible to use a coprocessor card or graphic processing unit (GPU). Coprocessor cards have tens to hundreds of processors to which sections of code could be off-loaded. GPUs comprise hundreds to thousands of simpler cores which execute instructions according to the Single Instruction Multiple Data (SIMD) model. In this model, all the cores execute the same instructions, but on different data sets.

V. Implementation

A. Reception and transmission of packets

Packets are received and transmitted using netmap ring structures. In the case of reception, application reads packet descriptors starting from the *head* pointer. *Head* pointer is incremented whenever a packet is read. When *head* pointer reaches *tail* pointer, synchronization is initiated through *ioctl* system call. Synchronization represents an exchange

of slots between netmap and NIC rings. After synchronization, application will receive a new set of slots for processing.

On the transmission side, the opposite procedure is performed. Application writes packets between *head* and *tail* pointers. When the *head* pointer reaches *tail*, synchronization is triggered. Through synchronization, empty slots from the NIC rings and filled slots from the netmap rings are exchanged.

B. Packet processing

From each packet, source and destination MAC addresses are extracted. If the destination MAC address is not appropriate, i.e. the packet destination MAC address does not match the port MAC address, this packet is discarded. Afterwards, it is determined if the packet has an IP header, and if all of its flags are set correctly.

IP lookup is performed based on the destination IP address. The result is information about the next hop device to which the packet should be forwarded and through which port. According to the next hop information, new MAC addresses will be obtained, and they will be inserted into the Ethernet header.

In order to avoid costly system calls, frequent function calls are avoided. Function calls cause huge overhead by putting current processor context and arguments on stack. Therefore, each function in the time-critical section of code should be implemented as inline. Additionally, packets are not copied at all throughout the application.

C. IP lookup procedure

IP lookup tables are often implemented as binary tries [8]. These structures have nodes that represent IP prefixes. The depth of node is the number of network bits in a prefix. If the route exists for a certain prefix, then, the corresponding node will have the route assigned. All subtries with the nodes without assigned routes can be deleted.

Lookup procedure starts from a root node in a trie. Destination IP address is examined bit by bit, starting from the MSB. If the bit is 0, algorithm will examine left child node, if it is 1, it will examine the right child node. If the examined node has been assigned a route, it will become a new longest prefix node. The procedure is repeated for the child node of the examined node that corresponds to the following IP destination address bit. Algorithm ends when there are no more child nodes to be examined. The result of the algorithm is the route of the longest prefix node.

The IP lookup based on binary trie might take a long time. In the worst case, 32 reads are necessary from the shared memory. For this reason, multibit trie structures are preferred [8]. Multibit trie structures are similar to binary tries, but each node has multiple children nodes whose number is denoted as *stride*. Usually, *stride* is power of 2. By using this technique, the maximal number of memory reads is reduced to 32/*stride*.

Search algorithm using multibit tries is similar, now in each step of the algorithm *stride* of bits is examined. With the increase of the *stride* length, the lookup speed increases, as well as the memory consumption. Usually, the *stride* length is a trade-off between memory size and lookup speed.

D. Parallelization

Netmap framework allows opening a separate file descriptor for each ring. One thread will be in charge of one Rx/Tx ring pair per each port. Thread will only perform synchronization for its own set of rings.

Parallelization is based on the RSS mechanism for forwarding packets from input FIFO into different Rx queues (rings). Each thread will be in control of one Rx ring from each port. Depending on the routing decision, thread will forward a packet to one of the available ports. Each thread will also control one Tx ring from each port.

We have designed two different parallelized IP lookup applications – the first one is parallelized by means of POSIX threads, at low level, and the second one is parallelized by means of OpenMP, at high level. Our goal was to compare which parallelization method is better to use for the applications that require high packet processing speeds.

1) POSIX threads application

In the first application, parallelization is implemented by means of POSIX threads. The first part of the program is the initialization procedure. Each thread is created through *pthread_create* function. Afterwards, all threads are initialized. Each thread is configured to be cancelled in a deferred way. Deferred cancellation means that the thread will cancel itself slightly later after the request of the main thread, in order to avoid data corruption. Threads will terminate themselves only in parts of the code that are not time-critical. As a part of thread cancellation, the clean-up routine will be executed in order to release allocated resources. Clean-up routine is pushed to the thread's private stack during the initialization.

The main task of each thread is packet routing, where each thread performs the procedures explained in sections B and C. All threads calculate the number of routed packets, and write the values to a shared memory. After the synchronization procedure, each thread calls the *pthread_testcancel* function to check whether it should terminate. Additionally, the main thread is calculating the total number of routed packets by adding the number of packets routed on all threads.

The user requests the termination of program through the SIGTERM signal. Each second, the main thread is checking status of this signal. If the termination of the program has been requested, the main thread will call *pthread_cancel* function. This function announces to the other threads that they should terminate themselves. The main thread will, then, wait for this condition blocked on *pthread_join* function. Afterwards, it will terminate itself.

2) OpenMP application

Pthreads are the low level parallelisation method which provides finer control of the behaviour of the threads. However, this API can be quite complex to use. It is much easier to parallelize the program at the higher level, in the OpenMP environment.

We have parallelized the program through the use of the parallel OpenMP sections (*#pragma parallel omp*), as each

thread performs essentially the same packet processing routine. The main thread is different, as it has to calculate the routing speed and check for the termination signal. This part of the code is isolated by the pragma *#pragma omp master*. Upon receiving SIGTERM signal, the main thread sets a shared variable, which should be examined later as a condition to exit the packet processing while loop. All the threads will terminate themselves at that point in code, thus eliminating the problem of possible data corruption.

E. Accessing routing table

Access to the shared variables must always be synchronized in order to avoid race conditions. For that purpose, pthreads offer low-level mutex and spinlock objects. Accessing shared data through locking objects in time-critical section of the code is not favourable as it inhibits the routing speed. Therefore, locking operations should be avoided in the parts related to packet processing, if possible.

Shared variables in OpenMP are accessed and modified through the lightweight atomic instructions. The critical sections were avoided as much as possible, as they introduce serial executions in the code. Critical section is necessary only in the part of the code related to the opening of file descriptors, which is acceptable as this procedure is not in the time-critical code section.

In the case of our parallelized IP lookup application, the shared resource is the IP lookup table. Accessing the lookup table from all threads is safe, because none of them is modifying its content. However, control plane routing protocols must change routing table on the fly. For this reason, accessing routing table from data plane threads must be prohibited during the routing table update by control plane. When the main thread receives a request for table modification, it will stop execution of all data plane threads, until the routing table update is finished.

VI. Testing Environment

Three server machines were used, with characteristics given in Table 1. The first server is used as a packet generator. It generates packets on two ports with 10 Gbit/s rate per port. The second server is a software router with the installed IP lookup application. Router has four ports. It is configured to forward all the traffic received on the first two ports to the remaining 2 ports. This configuration does not reflect real life scenarios, but it is suitable to test the limits of a software router. The third server is used as a packet sink, and to measure the speeds at which the routed packets arrive.

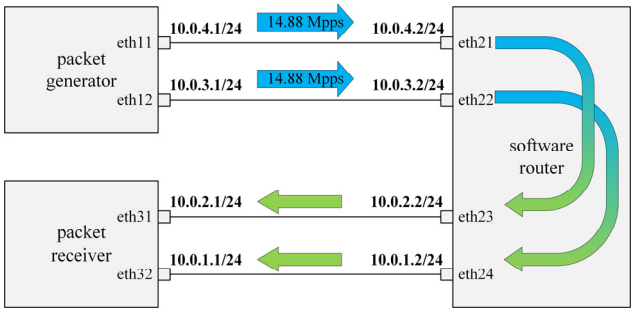

Fig. 3. Testing environment.

The first server generates TCP packets using DPDK packet generator [3]. Packets are generated with the highest packet rate necessary to saturate 10 Gbit/s links. The packet rate can be calculated according to (1):

$$pktRate = \frac{bitRate}{8 \cdot (pktSize + 20)} \quad (1)$$

where *pktRate* and *bitRate* are the packet and bit rates respectively, and *pktSize* is the packet size. The constant 20 comes from 20B of interpacket gap, preamble and start of frame delimiter in Ethernet frames. Maximal packet rate available on 10 Gbit/s link is 14.88 Mpps, which is achieved for the minimal size Ethernet frames of 64 B. In the case where two ports are used, this results in 29.76 Mpss in total.

TABLE 1: CHARACTERISTICS OF TEST MACHINES

	packet_generator	*packet_receiver*	*software router*
Processor	Intel Core i7-3770K	2x Intel Xeon E5-2620	Intel Core i7-3770K
# of Cores	4 per CPU	6 per CPU	4 per CPU
# of Threads	8 per CPU	12 per CPU	8 per CPU
Frequency	3.5 GHz	2.0 GHz	3.5 GHz
Cache size	L1 = 32K + 32K L2 = 256 KB L3 = 8 MB	L1 = 32K + 32K L2 = 256K L3 = 15M	L1 = 32K + 32K L2 = 256 KB L3 = 8 MB
RAM size	16GB DDR3 @ 1600 MHz	16GB DDR3 @ 1066 MHz	8GB DDR3 @ 1333 MHz
NIC	Name: AOC-STGN-i2S Controller: Intel 82599 Number of ports: 2 Speed: 10 Gbit/s per port	Name: AOC-STG-i4S Controller: Intel XL710-AM1 Number of ports: 4 Speed: 10 Gbit/s per port	Name: AOC-STG-i4S Controller: Intel XL710-AM1 Number of ports: 4 Speed: 10 Gbit/s per port
OS	CentOS Linux 7	Ubuntu 14.04	Ubuntu 14.04

Two different sets of tests were performed. In the first set, packets were generated on one port of the packet generator only. In the second set, packets were generated on both ports. For each set, tests were conducted for different packet lengths: 64 B, 128 B, 256 B, 512 B, 1024 B and 1500 B. For a given packet size, tests were repeated for different number of flows: 1, 2, 4, 8, 16, 32 and 64. Flows are separated according to the source TCP address.

Software router receives all the packets on the two ports, and forwards them to the other two ports, as depicted in Fig. 3. Software router has 8 logical cores, and also 8 input and output queues per port. IP lookup table is realized using multibit trie structures, with stride length equal to 4. IP lookup table is filled with 16777216 routes. This table does not reflect any practical situation, but it is useful for testing as it requires the most read-outs from memory per packet. Each of these routes specify server 1 as the next hop device.

VII. Results and Discussion

We have measured speeds of the IP lookup in Linux kernel, IP lookup netmap application parallelized with Pthreads, IP lookup netmap application parallelized with OpenMP, and the speeds the packet transfers from one port to another without any modifications. The last value shows

the maximal performance that the netmap framework may achieve. All measured values are compared with theoretically maximal packet rate which will saturate 10 Gbit/s links. Each test case is run for 10 seconds, and the average value of packet rate is calculated.

Fig. 4 shows packets rates of the IP lookup application parallelized with Pthreads, IP lookup application parallelized with OpenMP, IP lookup in Linux kernel, packet forwarding without packet modification in netmap (netmap bridge) and theoretically maximal packet rates in the case when one data flow is generated. In Fig. 5 packet rates are shown for the same applications and 64 data flows generated on one port. Packet rates are expressed in mega packets per second (Mpps).

From Fig. 4, we can see that the netmap bridge application is able to forward the packets with theoretically maximum packet rates. This confirms that the netmap itself is able to support such speeds. The performance of the applications parallelized by means of pthreads is almost identical to the OpenMP application, i.e. parallelization at higher level has not degraded routing performance.

The worst case situation for parallel applications is when one data flow is used, i.e. when no processing could be done in parallel. Maximal forwarding performance is around 3.8 Mpps, which is approximately four times lower than the maximal value. For the packets larger than 309 B, maximal achievable packet rates are reached. Parallelized applications perform the best when many data flows are used (Fig. 5). In this case, both parallel applications achieve maximal 14.88 Mpss rate.

The routing speed of the Linux kernel for one data flow is 1.06 Mpps, which is 14 times lower than the theoretically maximal value. For 64 data flows, the routing speed of Linux kernel improves, rising to 4.09 Mpps, which is 3.6 times lower than the maximum speed. By this, we have confirmed that Linux kernel indeed is not suitable to forward 10 Gbit/s traffic.

We also performed tests in which the data is generated on two ports. In Fig. 6, we have shown the forwarding speeds for the case where one data flow per port is generated. In Fig. 7, we have shown the forwarding speeds for 64 data flows per port. Tested applications are the same as in the previous graphs.

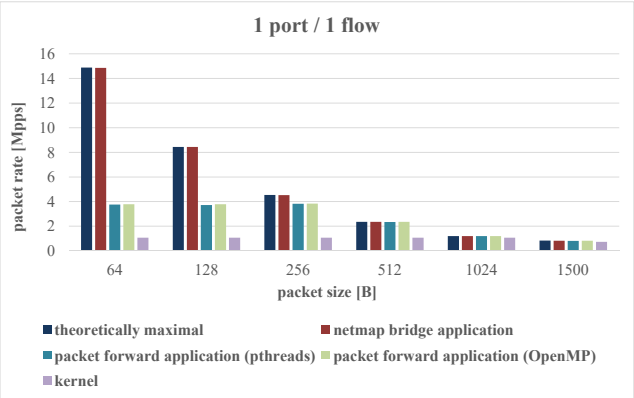

Fig. 4. Performance of IP lookup applications for different packet lengths and one data flow generated on one port.

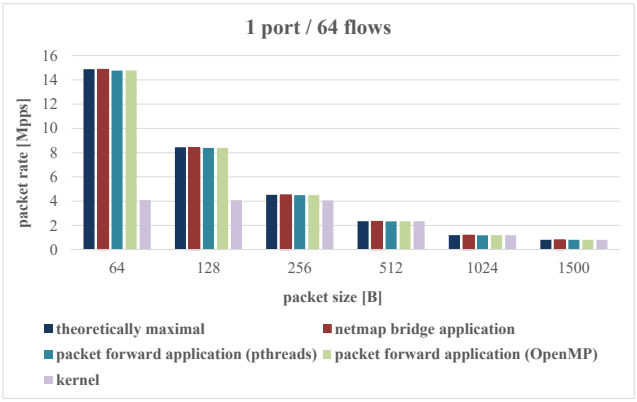

Fig. 5. Performance of IP lookup applications for different packet lengths and 64 data flows generated on one port.

In the case of one data flow per port (Fig. 6), it may be observed that the maximal forwarding speed of the netmap framework is 29.61 Mpps. This is slightly lower than the theoretically maximal packet rate for 64 B Ethernet frames for two ports, equal to 29.76 Mpps. The pthreads parallelized netmap IP lookup application achieves a forwarding speed of 7.38 Mpps, while the OpenMP parallelized application similarly achieves 7.49 Mpps.

Parallelized applications are approximately four times faster than the Linux kernel with the packet rate of 1.91 Mpps. Additionally, the IP lookup table in netmap application requires the maximal number of memory cycles, while the IP lookup kernel table is filled with static routes.

With the increase of packet length, it can be observed that the netmap performance approaches theoretically maximal values, while the performance of the IP lookup application and kernel IP lookup are the same as in the case of 64 B packets. For the packets longer than 512 B, the netmap IP lookup application reaches theoretically maximal packet rates. This is not the case for the Linux kernel, which cannot achieve maximal packet rates for any packet length.

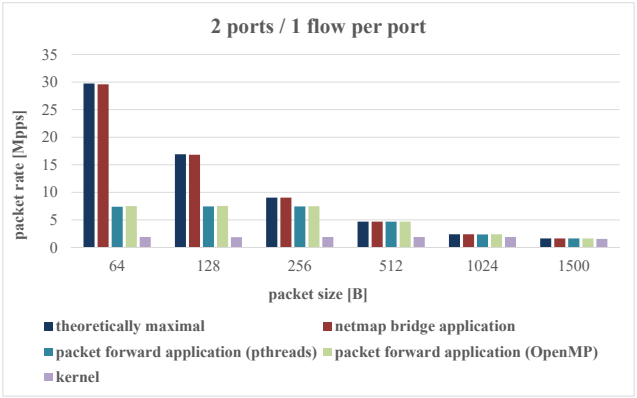

Fig. 6. Performance of IP lookup applications for different packet lengths and one data flow generated on each of the two ports.

In the case of the parallelized IP lookup (Fig. 7), the performance of the IP lookup in kernel and IP lookup in netmap are improved. For the 64 B packets, and 64 data flows per port, the forwarding speed of the netmap IP lookup parallelized by pthreads is 25.14 Mpps and the forwarding speed of the parallelized by the OpenMP is

25.26 Mpps. Once again, there are no observable differences between the applications parallelized in different manners. For the same test case, the forwarding speed of the IP Linux kernel is 4.05 Mpps, which means that the netmap IP lookup is approximately six times faster than the Linux IP lookup.

By leveraging the parallelization potential, i.e. RSS mechanism, we achieve more than three times better lookup speedup compared to the case when RSS is not used.

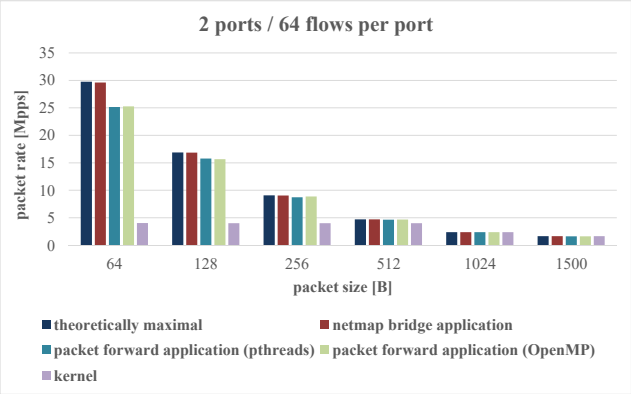

Fig. 7. Performance of IP lookup applications for different packet lengths and 64 data flows generated on each of the two ports.

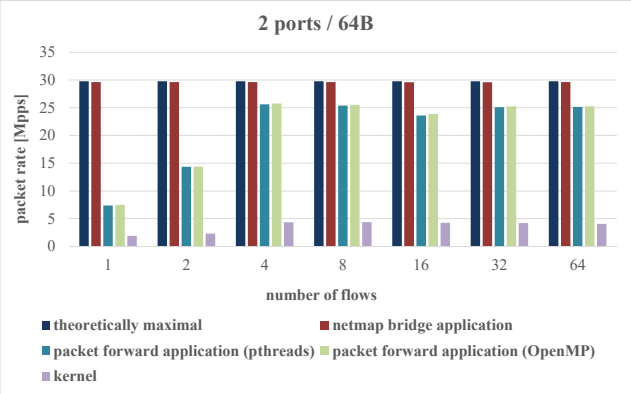

Fig. 8. Performance of IP lookup applications for different packet lengths and one data flow generated on each of the two ports.

We have concluded that the forwarding speeds are improving with the increase in the number of flows, thanks to the parallelization. In Fig. 8, we can see how packet rates scale with the increase in the number of flows per port. The limit is achieved for the number of flows per port equal to 4, both for the netmap applications, and the Linux kernel.

VIII. Conclusion

In this paper, we have shown one efficient realization of the IP lookup in user space by using fast I/O frameworks and parallelization. We demonstrated that the routing speed of the user space application is approximately four times better than the Linux kernel routing speed, in the critical case of minimal 64B packets. The results show that the parallelized application is more than three times better than the sequential one. Finally, we concluded that it is much easier to perform parallelization for packet processing applications on a higher level, without introducing performance penalty.

References

[1] E. Guillen, A. M. Sossa and E. P. Estupiñán "Performance Analysis over Software Router vs. Hardware Router: A Practical Approach," *Proceedings of the World Congress on Engineering and Computer Science*, vol. *2,* pp. *24-26*, 2012.

[2] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[3] Intel Corporation, "DPDK – Data Plane Development Kit," [Online]. Available: http://dpdk.org/. [Accessed 28.04.2016].

[4] 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.

[5] R. Buchty, V. Heuveline, W. Karl and J. Weiss, "A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators", *Concurrency and Computation: Practice and Experience*, vol. 24, no. 7, pp. 663-675, 2011.

[6] Blaise Barney, Lawrence Livermore National Laboratory, "POSIX Threads Programming", 17. June 2016. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/. [Accessed 07.07.2016].

[7] Intel Corporation, Networking Division, "Intel® Ethernet Controller X540 Datasheet", January 2016. [Online]. Available: http://www.intel.com/content/www/us/en/embedded/products/networking/ethernet-x540-datasheet.html. [Accessed 07.07.2016].

[8] M. A. Ruiz-Sanchez, E. W. Biersack and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," IEEE network, vol. 15, no. 2, pp. 8-23, 2001.

[9] M. Vesović, A. Smiljanić and M. Tomašević, "Speeding up IP lookup procedure in software routers by means of parallelization," *2016 24th Telecommunications Forum (TELFOR)*, Belgrade, 2016, pp. 1-4.