# MPLS Implementation for the Linux Kernel

Igor Maravić and Aleksandra Smiljanić
School of Electrical Engineering, Belgrade University,
Belgrade, Serbia
Email: {igorm, aleksandra}@etf.rs

***Abstract*—There is an increased demand for new functionalities of the network equipment. Smaller companies that develop such functionalities build their products based on the open-source solutions. *Linux* is currently the most popular open-source operating system and thus the majority of products are based on it. As MPLS is becoming indispensable in the network core, the need for more stable version of MPLS for *Linux* becomes evident. Thus, this work focuses on the improvement of MPLS implementation for *Linux* kernel. It presents architecture and performance of this implementation. Its performance is compared to the performance of IP packets forwarding and existing MPLS implementations.**

***Index Terms*—MPLS; Linux; performance evaluation;**

## I. Introduction

*Multi Protocol Label Switching* (MPLS) [1] represents a simple technology, which is today indispensable in network cores. MPLS protocol adds a 32-bit header between the second and the third network layer headers. Within these 32 bits, 20 bits are reserved for the label, 3 bits for the *Traffic Class* (TC) field, 1 bit for the *Bottom of Stack* (S) field and the remaining 8 bits are reserved for the *Time to Live* (TTL) field. A traffic

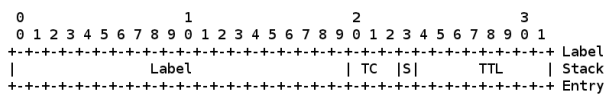


Fig. 1. MPLS Header

class carried by means of a MPLS packet is defined based on the TC field. Definition of an S field enables assigning of several labels to the packets. In the label grouping, only the lowest label, closest to the IP packet, has S bit set to 1. All other labels in the label stack have S set to 0. TTL field is used just like a TTL field of the IP packet.

Forwarding of packets in MPLS network is done on the basis of the labels, which have a local significance only. The first benefit here is a shorter time needed to define a next packet hop. In IP networks, the lookup currently represents a bottleneck in forwarding of packets. The transition from IPv4 to IPv6 protocol provides for the increase of addresses from 32 to 128 bits. As a consequence, the lookup, which has to be performed for each packet, becomes even more complex. The primary benefit of MPLS is seen in the fact that only the ingress router in MPLS network (LER - *Label Edge Router*) performs an IP lookup. Based on this lookup, it adds a label to the packet, and forwards it into the MPLS network. In other MPLS network routers (LSR - *Label Switching Router*), next hop decisions are exclusively made based on the label of packet, which is modified in every router, thus having only local significance and due to this it can be significantly shorter.

The second benefit of MPLS technology is that it provides an explicit traffic engineering (TE) [2]. It enables routing of specific traffic flows through paths that meet relevant quality of service (QoS) requirements and incur a more balanced traffic load on links. By enabling a more balanced use of links, it prevents the situation where there is congestion in one part of the network while other parts are feebly used. TE is most frequently enabled by establishing explicitly label switched paths (LSP). Explicit LSPs can be managed manually or by means of a routing protocol (RSVP-TE or CR-LDP). Depending on the needs, LSPs can be configured to route a different kind of traffic to the same destination by means of various defined paths. Routing paths are provided depending on the QoS needs for separate traffic flows. Attributes can be assigned to each LSP, such as a minimum delay, jitter and bandwidth, which it needs to meet.

This paper will present the work on the applicable version of MPLS for *Linux* [3]. The goal is to improve performance and utilization of the MPLS current version. Our work is under the GNU GPL license and may be accessed at [4]. In parallel to the development of this MPLS version, we have developed an improved version of *iproute2* utility suite, which can be accessed at [5]. This paper will show significant improvements of the new MPLS implementation compared to the previous solution.

Section 2 presents previous MPLS implementations. Section 3 describes the MPLS architecture in *Linux* kernel and our key contribution to the implementation. Section 4 analyzes the performance achieved by the improved MPLS, which is then compared to the performance of IP forwarding implemented in *Linux*, and to the performance of the implementations provided by other authors. Finally, a conclusion and guidelines for future work are given.

## II. Related Work

Nowadays, new companies, emerging on the market, become increasingly dedicated to the development of products based on open-source software solutions. This is how they save time for the development of new products. Besides saving their time and money, they make a contribution to the development of open-source software solutions. Users also benefit from this way of work by getting ready flexible solutions that can be easily upgraded with the appearance of new technologies,

MPLS being one of those technologies. As MPLS has recently become indispensable in core networks, there is an increasing need for operational and stable implementation of MPLS into operating systems (OS).

In view of the fact that *Linux* is the most popular open-source OS, the greatest need for MPLS implementation lies precisely there. The development of MPLS version for *Linux* was started by James Leu in 1999. At the same time, LDP protocol on *Quagga* software router was developed [6]. Unfortunately, LDP was never completed and has limited functionalities.

Of all open-source OSs only *NetBSD* OS, starting from version 6.0, officially incorporates MPLS in its network code, to the best knowledge of authors [7]. Together with this implementation, *NetBSD* also implements the LDP routing protocol for the distribution of MPLS labels [8].

Currently, there are some efforts to incorporate MPLS into the *FreeBSD* code [9]. This implementation is not officially supported in *FreeBSD* and is not fully functional. Simultaneously, LDP protocol is developed for the Bird software router [10]. LDP protocol for Bird is currently at beta phase. Bird, apart from operating on *FreeBSD*, can operate on *Linux* and *NetBSD*, as well. In order to be able to use this LDP implementation on other OSs as well, it would be needed to add a code for communication with the MPLS module of the other OSs.

Last year, MPLS was experimentally implemented within the *OpenFlow* [11], thanks to efforts made by *Ericsson* researchers [12]. *OpenFlow* represents a new open protocol that defines the communication of the network controller with the switch core. However, this implementation is still at an early experimental stage of development. Route entries in the MPLS routing table (LFIB - *Label Forwarding Information Base*) can be written only by means of *xml* files. The other problem is that the controller cannot process packets coming with several labels. Authors have tested the solution using a PC for LFIB maintenance, and a *NetFPGA* card for packet forwarding. Label manipulation functions in LFIB are taken from the existing MPLS implementation for *Linux*. The authors made a Python daemon, which was used for the periodic refresh of LFIB on the *NetFPGA* card, using the information from LFIB in the *Linux* kernel. Forwarding of MPLS packets through the *NetFPGA* card can achieve the maximum port utilization, as expected.

Relying on the MPLS implementation in *OpenFlow*, Stanford researchers have implemented a functionality of MPLS control protocols within NOX network OS [13]. NOX represents an OS that supports centralised network operation. This work does not aim to solve the problems that existed in the MPLS *OpenFlow* implementation. Functionalities of MPLS control protocols, such as RSVP-TE and LDP, have been successfully demonstrated in a virtual domain, on a *Mininet* platform.

Another MPLS implementation was presented in [14]. This implementation is based on a *Click Modular Router*. Basic operations of swapping, popping and pushing labels were defined within the implementation. The implementation was successfully tested. Unfortunately, unlike the previously presented implementations, this one is not an open-source implementation.

The MPLS version presented in this paper is a scalable, flexible and expandable solution. Unlike other implementations, our implementation allows for the manipulation of the TC field. As an additional advantage, it is easy to map values of DSCP field from the IP header into the TC field. This implementation has a unique capability of making a recursive query in the MPLS routing table. The advantage over the *OpenFlow* implementation is an easiness of entering/modifying/deleting labels from routing tables, owing to the fact that our implementation manipulates the data in LFIB simply by means of a command line and not *xml* files. The other advantage is the ability to manipulate packets that have several labels assigned. As we will show, our implementation can achieve higher speeds than some of the previous solutions.

A licensing method is inherited from the original MPLS implementation for *Linux*. This implementation of ours can be located under the GNU GPL license. This means that all interested parties are free to use it and adjust it to their needs without any remuneration.

## III. MPLS Implementation

This section is aimed at presenting the implementation architecture and the key improvements achieved with respect to the original version of MPLS in *Linux*. Section III-A shows the MPLS implementation architecture, while section III-B presents our key improvements of the original version.

### A. Architecture of MPLS Implementation in Linux

MPLS implementation is devised to be integrated in the network part of *Linux* code. Just like the MPLS header is

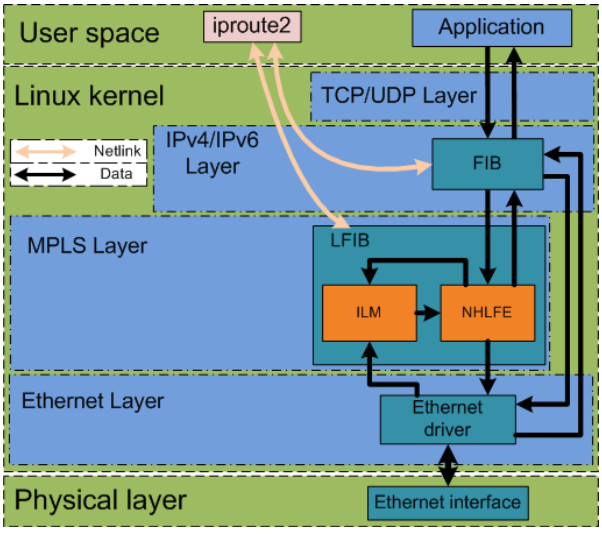


Fig. 2. MPLS implementation in *Linux* kernel

located between the second and the third network layer, the MPLS code is located between the IP code and the code

managing the functions of the second network layer. This is shown in Fig. 2.

In the implementation itself, LFIB is split into two parts [1]. The incoming part of LFIB is called ILM (*Incoming Label Map*) and outgoing part is called NHLFE (*Next Hop Label Forwarding Entry*). Entries in the NHLFE table keep MPLS instruction sequences that are executed over the packet. An interface between IP and MPLS layers is realized via the NHLFE table. When a packet is forwarded from IP layer to MPLS layer, it needs to know to which entry in the NHLFE table it should be forwarded to.

A sequence of instructions, to which each NHLFE entry must point, is presented with the linked list of elements. Each element is related to only one function performing packet manipulation. Basic instructions are *pop*, *push*, *send* and *peek*. Label removal from MPLS packet is done by means of a *pop* instruction. Adding MPLS labels to packets is done by means of a *push* instruction. As for the *send* instruction it serves for the sending of packets to the next hop. When its last label is taken off, a MPLS packet can be forwarded to the *send* instruction and then to the next router without returning to the IP layer. This is called MPLS penultimate hop popping (PHP). By using MPLS PHP, resources of the router and its next-hop router are conserved as they do not have to do both the MPLS and the IP lookup, but only one of them. The *peek* instruction is performed in two ways. In case all labels are removed from the packet, the corresponding element forwards the packet for processing to the IP layer, and in case some labels remain, the packet is returned to the MPLS layer for additional processing. The *swap* instruction is not specially defined. It is presented with the combination of *pop* and *push* instructions. Apart from basic instructions, instructions of mapping TC field, mapping of TC field depending on DSCP field and mapping of DSCP field depending on TC field are also defined.

MPLS lookup is done through the ILM table. Entries of the ILM table are uniquely defined by the label and *label space*. The MPLS *label space* is a domain where label definitions must be unique. The same labels, however, may be defined in the system, but they must belong to different *label spaces*. The MPLS *label space* may be defined for an individual interface or an interface group. Each ILM entry must contain a pointer to the NHLFE entry, so that it could forward the received MPLS packet for further processing. When the lookup algorithm finds a relevant ILM entry based on the label from the packet header and *label space* of the port from which the packet has arrived, the packet is forwarded to the relevant NHLFE table entry. An option to forward the packet to different NHLFE entries depending on the TC field values has been implemented as well. Fig. 3. shows a flow of packets through MPLS code while performing a MPLS *swap* instruction.

The *iproute2* utility suite has been expanded to enable the MPLS administration. The communication between processes in kernel and *iproute2* programme is done exclusively by means of *Netlink* protocol. This is very important, since the *ioctl* communication is outdated and overall communication

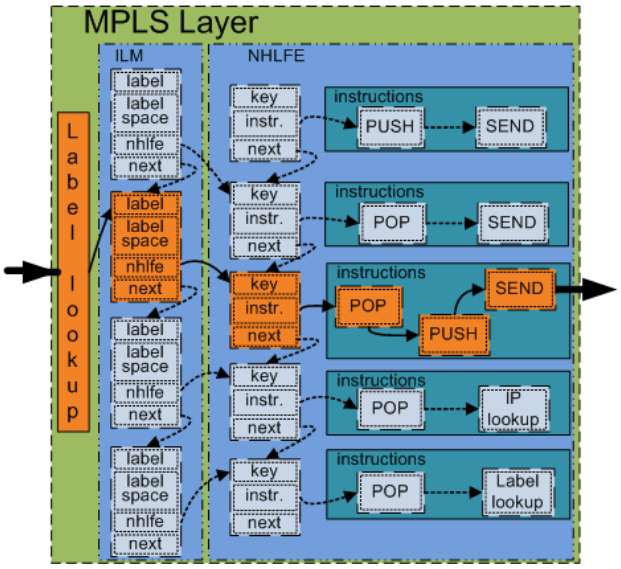


Fig. 3. Flow of packets through MPLS subsystem at MPLS *swap* function

with kernel tends to be performed via *Netlink* protocol.

### B. Improvements of MPLS Original Version for Linux

The original version of MPLS, which was the starting point for this work, was good conceptually, but its implementation still has many pending problems. Beside the pending bugs, the last MPLS version was designed for an outdated version of kernel. The current version of the *Linux* kernel is 3.2.1. while the last version for which MPLS was written is 2.6.35. Big discrepancies existed between those two versions in respect of a network code that needed to be overcome.

First, we resolved the system failure problem, which occurred when kernel was compiled to execute sanity checks. Kernel *panic* appeared at the point of establishing a link between the IP lookup and NHLFE entry. The old method of establishing resulted in a stack overflow. A solution was found in modifying the type of variable that keeps this link from the structure type to the type of a pointer to the structure.

Secondly, execution of MPLS instructions is accelerated. It is defined that instructions are allocated in the continuous memory space. All functions corresponding to the instructions were optimized and the code is shortened to the maximum extent for all of them. In course of optimizing instructions, a special attention was given to use already defined functions for manipulating the *sk_buff* structure. In *Linux*, *sk_buff* represents an abstraction of data packets. All packet operations are managed in kernel trough this structure.

Then, forwarding of packets to higher and lower layers is modified to use already existing functions from the network code. Forwarding of packets to another network layer is realized in the same way as forwarding of packets from the IP layer to another network layer. Forwarding of packets to the IP layer is solved fully in line with the forwarding from another layer to the IP layer. Thus, the communication with

other network layers was optimized and transfer of code to the future kernel versions was facilitated.

*Simple Network Management Protocol* (SNMP) statistics was added. The statistics has been realized in line with RFC 3813 [15]. It enables the system to follow up the number of sent, received and rejected MPLS packets. Entire statistics is written in the */proc/* file system. In our modified MPLS version, there is a possibility to perform an unlimited number of MPLS instructions over each packet, while the number of instructions was limited to 8 in the original version.

Finally, communication over *Netlink* is modified. Now all writings/modifications/deletions of data in LFIB, as well as modifications of *label space*, are updated in user space. A lot was done on facilitating the MPLS administration through an *iproute2* programme. The most important change is enabling modification of already existing entries in LFIB.

## IV. Performance Evaluation

The aim of this section is to present achieved performance of the software router with our MPLS implementation. Section IV-A describes the methodology of performing experiments and used equipment. Section IV-B presents the results of experiments. Section IV-C compares the performance of the MPLS software router with the performance of the existing MPLS implementations.

### A. Experimental Setup

The router consisted of an ordinary home PC with a M2N68 PLUS ASUS motherboard, AMD *Athlon* II X2 240 processor at 2.8GHz, 4GB DDR2 RAM memory at 1066MHz and with 3 *Intel* network interface cards (NIC). The fourth NIC used in experiments was integrated on the motherboard. *Intel*s NICs used *e1000e* driver, and integrated NIC used a *forcedeth* driver. The PC had an installed OS *Ubuntu Server* 11.10 with compiled MPLS *Linux* kernel.

All the experiments were done for various packet sizes, ranging from 60B to 1500B. Bidirectional flows were observed in all tests. This means that the traffic was sent and received at the same time on all NICs. All four NICs were given a maximum load of traffic, for all packet sizes. In order to analyse software router performance in greater detail, daemon *irqbalance* [16] was off in all the experiments. CPU affinity for interrupt requests (IRQ) was set in the files */proc/irq/*, so that one CPU core was used by two NICs. During the measurements, it was important to note whether NICs, having exchange of packets, are processed by the same CPU or not. Thus, two cases were observed. In the first scenario, packets passing through the router were using NICs served by the same CPU. In the second scenario, packets were using NICs served by different CPUs.

Router's performance was analysed for IP traffic forwarding, MPLS *swap* function, combination of MPLS *pop* and *swap* functions, MPLS *push* function and MPLS PHP. As the IP module of the *Linux* kernel code was tested in detail and optimized to the maximum extent, IP forwarding performance was taken as a benchmark to which we compare the performance of our MPLS implementation. IP packets were generated and sent to the router's NICs in order to test the IP traffic forwarding through the software router. Such packets would be carried through the IP module of kernel and were forwarded to an interface specified in the IP lookup table.

IP packets were generated and sent to the router in order to test MPLS *push* function. These packets were mapped to their MPLS labels which were added to them, and then, they were forwarded to one of outgoing ports. Packets with MPLS labels were generated at the ingress, in order to test MPLS *swap* function. New labels were calculated and assigned to those packets. Then, they were forwarded to one of outgoing ports.

Packets with MPLS labels were generated in order to test MPLS PHP function too. The packet label was removed and the packet was then carried based on the entry in the NHLFE table. A combination of MPLS *pop* and *swap* functions was tested to simulate packet processing in a *context-specific label space* [17]. It was tested by generating packets labelled with two MPLS labels at the ingress. Firstly, the upper label was removed from the packets and then a *swap* function would be recursively performed for the lower label.

Throughputs were directly measured on the software router, by means of lightweight programmes *bmon* [18] and *mpstat* [19]. In the tests, IP and MPLS lookup tables contained only routes necessary for the tests. Simulations were made for each packet size during 30s, while calculating an average value of the resulting throughput.

### B. Results

Fig. 4. and Fig. 5. show the IP and MPLS throughputs when packets use a single CPU, and when they use both CPUs, respectively. From these figures, it can be concluded that there is no significant difference between the speeds of MPLS and IP traffic forwarding. These results show the excellent

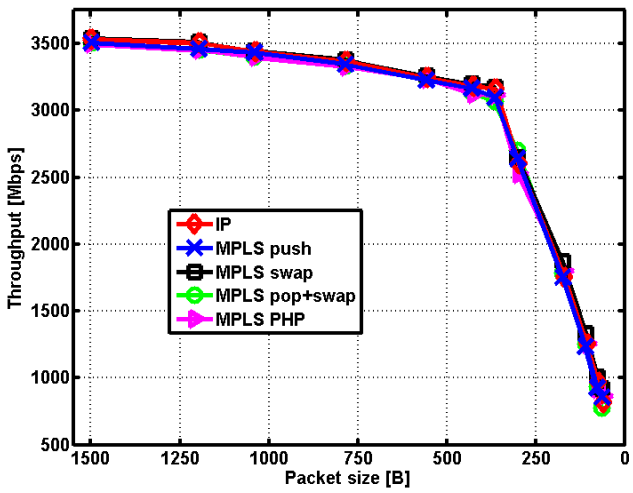


Fig. 4. Packet throughput - each packet uses a single CPU

performance of our MPLS implementation, because the IP module of *Linux* kernel was tested in detail and optimized

to the maximum extent. Our implementation is, therefore, a significant improvement of the original version of MPLS, which had much worse performance in comparison to the IP forwarding. Please note that small IP lookup tables were used, so the IP lookup was not a bottleneck.

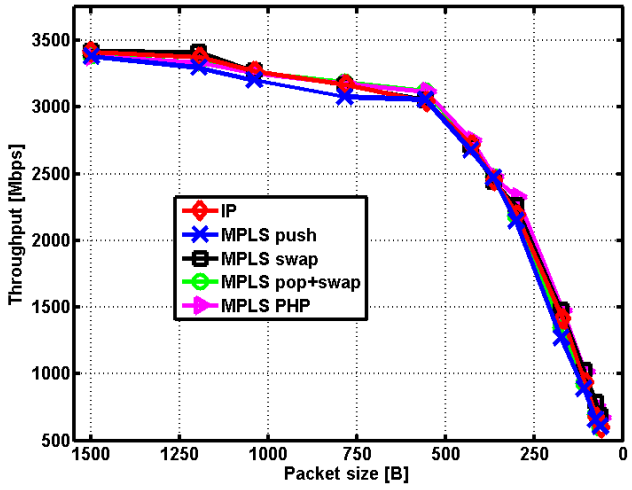


Fig. 5. Packet throughput - each packet uses both CPUs

In both cases, the MPLS *push* function has the worst performance of all tested methods of packet forwarding. This was to be expected, since in course of MPLS *push* function testing, the packet went through an entire processing of IP stack to finally reach the MPLS part of the code, where MPLS *push* is done.

However, the variation between forwarding of packets for different MPLS functions is minimal in both cases. This is due to the fact that MPLS functions, corresponding to different MPLS instructions, are shortened to the maximum possible extent so that the speeds of their executions are optimized.

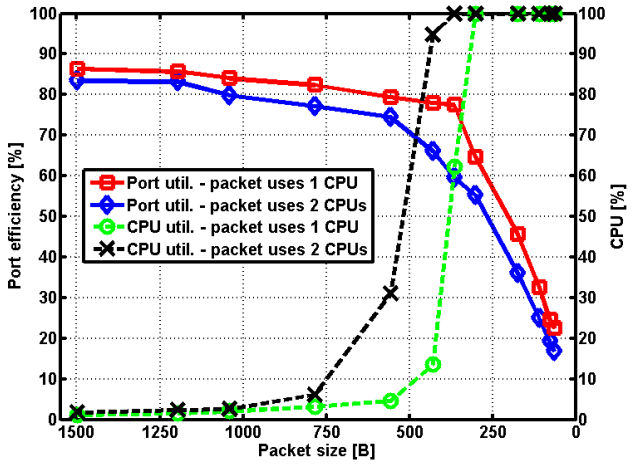


Fig. 6. CPU and port utilization for MPLS swap

As expected, throughput is higher when packets are processed only by one CPU. This is due to the fact that additional processing time is required to carry the packets processed by one CPU to the other CPU, as can be observed in Fig.6.

Fig. 6. show the port and CPU utilization in the case of the *swap* function. When packets pass two CPUs, apart from having the worse packet forwarding speed, there is an increase in the use of CPU resources. Also, it is evident that as soon as the utilization of CPU resources begins to fall below 100%, packet flow saturates. When it happens, a bottleneck is not any more the processing power of a CPU but the capacity of NICs. Since the tests were done with NICs of the average quality, the maximum throughput does not exceed 90% of their maximum possible value.

### C. Comparison With Existing Implementations

In this section, we will compare the performance of our implementation with the performance of the original MPLS implementation in *Linux* [12] and the performance of the MPLS *Click* implementation [14].

For performance evaluation of the original MPLS implementation in *Linux* [12], authors used the latest available MPLS version at the time, written for 2.6.35 *Linux* kernel. These tests were performed for only one bidirectional traffic flow. The comparison is shown in Fig. 7.

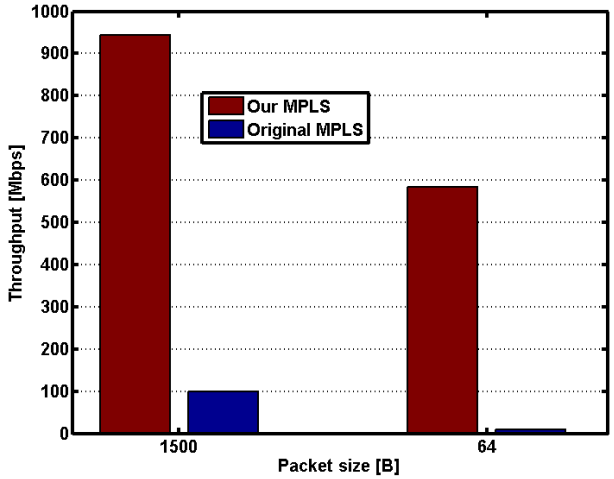


Fig. 7. Packet throughput compared with original MPLS version [12]

An improvement achieved by our implementation is clearly evident. Fig. 7. shows that packet forwarding is about 9 times increased for the forwarding of the largest packets and approximately 55 times for the smallest packets, in the case of our MPLS implementation.

Authors of the *Click* implementation performed testing for two scenarios. Scenario A presented the case where the traffic coming to eight 1Gbps ports was forwarded to one 10Gbps port. In scenario B, the traffic coming to four 1Gbps ports and to one 10Gbps port, was then forwarded to a 10Gbps port. In theory a maximum throughput for scenario A is 8Gbps, and 10Gbps for scenario B. Both scenarios are tested for two ways of IRQs allocation. The first method of IRQs allocation has all IRQs from NICs processed by one CPU. It is called a *no affinity* case. In the second case, IRQs from NICs are allocated evenly to all CPUs. This case is called a *full affinity* case. As

the *full affinity* case showed significantly better results, our implementation was compared with these results only.

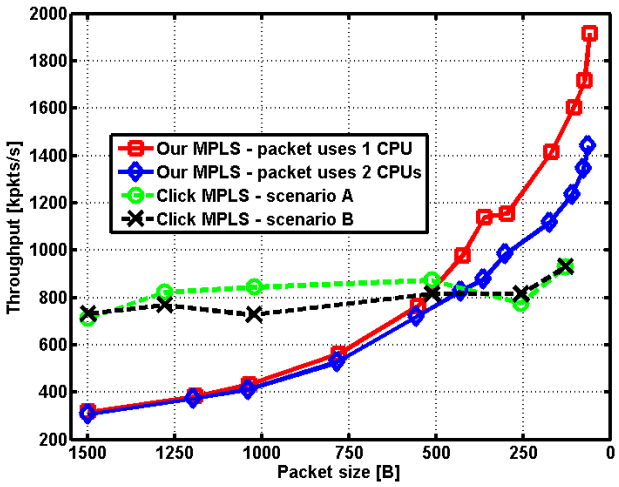


Fig. 8. Packet throughput compared with *Click* implementation [14]

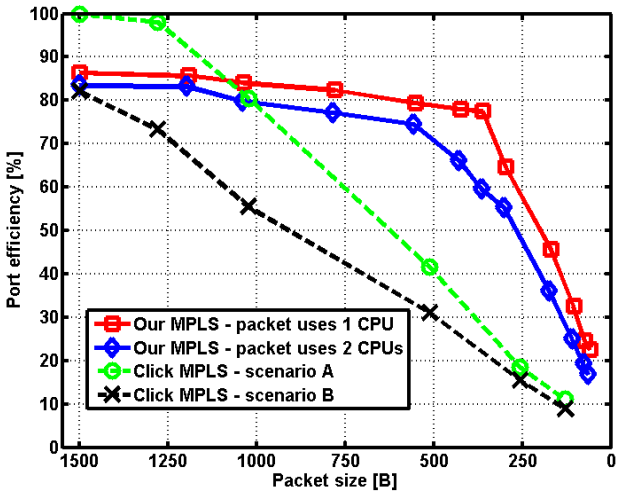


Fig. 9. Port efficiency compared with *Click* implementation [14]

Comparing the performances of the *Click* and our MPLS implementation (Fig. 8. and Fig. 9.), it can be seen that our implementation is by far better when CPU is being a bottleneck for packet processing (Fig. 8.). For the smallest packets, our implementation has about 1,7 times greater number of processed packets in a second! Here, it is worth noting that our test was made on a *DualCore* processor, while the *Click* MPLS test used two *Intel Xeon QuadCore* processors. It should be also noted that a recommended price of *Intel Xeon* processor, used by authors, exceeds the overall price of our test engine. When the quantity of processed packets starts to depend on NICs limit, the *Click* implementation achieves a greater throughput, because it used more NICs. However, if we measure the performance more fairly, by the port efficiency, our implementation achieves better results for almost all packet sizes as shown in Fig. 9.

## V. Concluding Remarks

In view of the fact that MPLS protocol is becoming indispensable in the network core, and that software routers based on *Linux* OS are in expansion, there is a growing need for stable implementation of MPLS for *Linux*. This paper presents an improved implementation of MPLS for *Linux*. The performance analysis showed a significant improvement of our new version over the original version of MPLS for *Linux*. Our implementation can support 9 times greater throughput for the largest packets, 1500B long, and approximately 55 times greater throughput for the smallest, 64B long, packets.

Our future work will include additional improvements of the MPLS implementation. Apart from the work on the MPLS implementation itself, an implementation of RSVP-TE protocol as a part of the software router is planned.

## Acknowledgement

This work is funded by the Serbian Ministry of Education and Science, and companies *Informatika* and *Telekom Srbija*.

## References

[1] E. Rosen, A. Viswanathan, and R. Callon, "*Multiprotocol Label Switching Architecture,*" RFC 3031, Jan. 2001.
[2] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell, and J. McManus, "*Requirements for Traffic Engineering Over MPLS,*" RFC 2072, Sep. 1999.
[3] J. R. Leu, "*MPLS for Linux,*" *http://mpls-linux.sourceforge.net/*, accessed Jan. 2012.
[4] I. Maravić, "*Improved MPLS for Linux,*" *https://github.com/i-maravic/MPLS-Linux*, accessed Jan. 2012.
[5] I. Maravić, "*Improved MPLS enabled iproute2,*" *https://github.com/i-maravic/iproute2*, accessed Jan. 2012.
[6] P. Jakma, et al. "*Quagga Routing Suite,*" *http://www.quagga.net/*, accessed Jan. 2012
[7] M. Chelaru, "*NetBSD Kernel Interfaces Manual - MPLS(4),*" *http://netbsd.gw.com/cgi-bin/man-cgi/man?mpls+4+NetBSD-current*, accessed Jan. 2012.
[8] M. Chelaru, "*NetBSD System Manager's Manual - LDPD(8),*" *http://netbsd.gw.com/cgi-bin/man-cgi?ldpd+8+NetBSD-current*, accessed Jan. 2012.
[9] A. Chernikov, "*MPLS for FreeBSD,*" *http://freebsd.mpls.in/*, accessed Jan. 2012.
[10] O. Filip, et al., "*The BIRD Internet Routing Daemon,*" *http://bird.network.cz/?index/*, accessed Jan. 2012.
[11] N. McKeown, et al. "*OpenFlow: enabling innovation in campus networks,*" SIGCOMM Computer Communication Review, vol 38(2), pp. 79-92, Apr. 2008.
[12] J. Kempf, et al. "*OpenFlow MPLS and the open source label switched router,*" 23rd International Teletraffic Congress (ITC), pp. 8-14, Sep. 2011.
[13] A. R. Sharafat, S. Das, G. Parulkar, N. McKeown, "*MPLS-TE and MPLS VPNs with OpenFlow,*" ACM SIGCOMM, Aug. 2011.
[14] R. Vilalta, R. Munoz, R. Casellas, and R. Martinez, "*Design and performance evaluation of a GMPLS-enabled MPLS-TP/PWE3 node with integrated 10Gbps tunable DWDM transpoders,*" IEEE 12th International Conference on High Performance Switching and Routing (HPSR), pp. 236-241, Jul. 2011
[15] C. Srinivasan, A. Viswanathan, and T. Nadeau, "*Multiprotocol Label Switching (MPLS) Label Switching Router (LSR) Management Information Base (MIB),*" RFC 3813, Jun. 2004.
[16] "*IRQbalance daemon,*" *http://irqbalance.org/*, accessed Jan. 2012
[17] R. Aggarwal, Y. Rekhter, and E. Rosen, "*MPLS Upstream Label Assignment and Context-Specific Label Space,*" RFC 5331, Aug. 2008.
[18] T. Graf, "*bmon - Bandwidth Monitor,*" *http://www.infradead.org/tgr/bmon/*, accessed Jan. 2012
[19] S. Godard, "*Linux User's Manual - MPSTAT(1),*" *http://www.linuxcommand.org/man_pages/mpstat1.html*, accessed Jan. 2012