Received June 7, 2020, accepted June 27, 2020, date of publication July 1, 2020, date of current version July 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3006370

Multicast Load-Balanced Birkhoff-Von Neumann Switch With Greedy Scheduling

SRDJAN DURKOVIC AND ZORAN ČIČA, (Member, IEEE)
School of Electrical Engineering, University of Belgrade, 11120 Belgrade, Serbia
Corresponding author: Zoran Čiča (zoran.cica@etf.bg.ac.rs)

ABSTRACT Internet traffic is still exhibiting an exponential growth. This exponential growth will certainly be continued given the Internet of Things (IoT) and Internet of Everything (IoE) predictions regarding the number of devices connected to the Internet in the near future. Also, many popular multicast services such as IPTV, distance learning, content distribution, distributive interactive gaming, collaborative computing and others are rapidly increasing amount of the Internet multicast traffic, thus, significantly contributing to the Internet traffic growth. The routers are typically designed to cope with unicast traffic. Multicast traffic can negatively impact performance of such routers and cause significant degradation of overall network performance. Hence, due to increasing importance and increasing amount of multicast traffic, there is a great need for a scalable switch architecture that efficiently forwards both unicast and multicast traffic. In this paper, we propose a novel scalable, efficient and frugal multicast switch architecture based on load balanced Birkhoff-von Neumann switch with greedy scheduling that achieves stable performance even at very high traffic loads. The proposed switch is compared to other popular multicast switch solutions. Comparison shows that our proposed multicast switch architecture outperforms other solutions in all tested common traffic scenarios at the most critical (highest) traffic loads.

INDEX TERMS Packet switching, multicast, high speed networks.

I. INTRODUCTION

Given the IoT and IoE trends, the number of devices connected to the Internet will be enormous in the near future [1]. Also, new Internet services are continuously emerging while attracting large number of users and generating large amounts of traffic. Among them, multicast services like IPTV, distance learning, content distribution, database replication and others are gaining a lot of attention and popularity [2], [3]. Therefore, the Internet traffic exponential growth trend will be continued [4]. Although, inter-domain multicast is still not widespread [5], [6], ISPs and data centers are using multicast for more efficient content delivery [6]–[8]. ISPs especially use the multicast for intra-domain video content delivery [6], while data centers use the multicast for various applications like streaming telemetry, replicated state machines, publish-subscribe systems, database replication that benefit from the multicast support in data centers [7]. Thus, the share of multicast traffic in the overall Internet traffic is not insignificant anymore and, nowadays, routers need to efficiently forward multicast traffic. At the core of a router, a packet switch is responsible for the traffic forwarding. The performance of a packet switch has crucial impact on overall router performance. However, most of the routers are based on unicast packet switches designed to efficiently cope with a unicast traffic. Usually, these routers have very poor performance when serving a multicast traffic. Given the Internet traffic growth and multicast traffic share increase, it is important to develop packet switch architectures that can efficiently support both unicast traffic and continuously increasing multicast traffic. The packet switch architecture needs to be very scalable in order to support high throughputs and large number of ports.

Input queued (IQ) packet switch architecture is very popular because it is significantly more scalable than the output queued packet switches. Many unicast IQ packet switches have been proposed in the literature [9]–[12]. Multicast extensions of unicast IQ switches can be divided in two main groups: 1) adjustment of existing unicast switch [13]; 2) introducing multicast capable switch fabric combined with IQ switch based scheduling algorithms [2], [3], [14]–[16]. However, the main problem of the IQ based approach is

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

the random switch fabric configuration pattern. The random pattern requires fast configuration calculations which can represent serious problem when a switch with large number of ports needs to support very high throughput such as 100Gbps throughput and beyond.

The load balanced Birkhoff-von Neumann (LB-BvN) switch architecture avoids the problem of random switch configurations, by using the deterministic switch fabric configurations [17]–[24]. LB-BvN switches exploit the fact that the deterministic switch fabric configuration is possible and simple for the uniformly distributed traffic. There are two switching stages with deterministic switch configurations. The first switching stage makes the incoming traffic as uniform as possible and the second switching stage just forwards the traffic to corresponding output ports. Because the switching pattern is deterministic, the LB-BvN solutions achieve great performance for the unicast traffic. However, the LB-BvN based solutions for the multicast traffic are not significantly investigated in the literature so far, even though they represent very promising multicast switch solutions [25].

In this paper, we propose a novel multicast architecture that is based on the unicast load balanced Birkhoff-von Neumann switch with greedy scheduling (LB-BvNGS) [24]. In order to support multicast traffic, we use multicast controller to build binary multicast trees in order to use simple unicast LB-BvNGS architecture for multicast packets. Each multicast flow has a corresponding binary multicast tree. Binary multicast tree comprises the input port at which the multicast packets are received and corresponding destination output ports to which the packets are forwarded. The input port that receives the multicast packets is the root node of the tree. Multicast packets are forwarded as unicast packet copies to destination output ports through the multicast tree. Since multicast tree is binary, at each non-leaf node of the tree at most two packet copies are created which limits the problem of potential input port overload with additional packet copies created from the original multicast packet. Hence, when a multicast packet arrives at the input port, it is forwarded to corresponding destination output ports by traversing all the links in the tree in downstream direction. One or two packet copies are created at every non-leaf node and forwarded to its children nodes. Note that the unicast LB-BvNGS is non-blocking with internal link speedup of two [24]. We show that our proposed multicast architecture is also non-blocking but with the internal link speedup of five due to additional packet copies that create additional traffic at the input ports in the worst case. We believe that this property is acceptable, given nowadays technology. Also, we show that in typical traffic scenarios even without internal link speedup our proposed multicast switch achieves high performance. The multicast controller is built in control plane and complies to software defined networking (SDN) paradigm. Thus, the routers that use the proposed multicast switch can be easily integrated in the SDN based network architectures.

This paper has a few contributions. A novel multicast switch architecture that is proposed is very scalable and achieves great performance even under very heavy traffic loads thanks to the fact that it is based on the efficient unicast LB-BvNGS packet switch architecture. Overall complexity of the proposed multicast switch architecture is low. The proposed switch is non-blocking for any admissible traffic scenario if internal link speedup of five is used. Also, speedup of switch configuration is not required for the non-blocking property. Note that most of the other existing multicast solutions do not prove non-blocking property for any admissible traffic scenario, but only show the performance for the typical traffic scenarios. However, even if the speedup is not used, the performances are still good for the common traffic scenarios. The multicast controller is built in the control plane and can be used in SDN based network architectures.

The rest of the paper is organized as follows. Section II covers related work. In section III, we give a detailed description of our proposed multicast load-balanced Birkhoff-von Neumman switch with greedy scheduling. Special attention is given to the multicast controller in this section. In section IV, we compare our solution to the other popular multicast schemes. Section V concludes the paper.

II. RELATED WORK

Several switch architectures with support for the multicast traffic have been proposed so far. Typically, the packet switches are optimized for the unicast traffic. However, due to the increase of the multicast traffic, the efficient multicast support has become an important feature of the modern packet switches. Most of the multicast solutions are based on the multicast support extension of the existing unicast packet switches. In this section, we give overview of the multicast switch solutions that have been proposed in the literature.

Many popular multicast solutions represent adjustment of existing unicast input queued (IQ) switches. IQ switches store incoming packets in buffers at the input ports. IQ switches are controlled by a scheduler that manages packet forwarding through the switch from input ports to corresponding output ports. Scheduler is responsible for calculation of connection patterns between input and output ports or, in other words, scheduler is responsible for the switch configurations. Some IQ switches, such as iSlip [9], [10], implement a central scheduler, but the central scheduler can limit the switch scalability. The pattern of IQ switch configurations is random in terms that switch configuration pattern heavily depends on the current traffic conditions. In the case when IQ switch needs to support high throughput, the switch configuration pattern must be calculated very fast, but the central scheduler works with N^2 entries which limits the scalability in terms of number of ports and supported throughput, where N is the number of input/output ports. Also, many centralized schedulers require multiple iterations in the switch configuration calculation process and that property has negative impact on scalability in term of the supported throughput. However, single iteration schedulers can overcome the problem of multiple iterations in the calculation process, but the problem of N^2 entries remains [11].

Distributed schedulers can be used to overcome the scalability problem of centralized schedulers by distributing the switch configuration calculation to all input ports. One such solution is the sequential greedy scheduler [12]. Here, a scheduling process is distributed across all input ports. Each input port schedules packets for some future configurations. For example, port i schedules one of its packets for time x, and at the same moment input port $i+1$ schedules one of its packets for time $x-1$. In the next iteration, the input port $i+1$ schedules one of its packets for time x. In this way, multiple switch configurations are calculated at the same time across all input ports using the pipeline that comprises chain of the input ports. Distributed scheduler at each input port works with only N entries because the scheduler is distributed across N input ports.

The aforementioned unicast IQ switch architectures can be adjusted to support multicast traffic. One way is to adjust the multicast traffic to unicast switch. Naive way to adjust the multicast traffic is to create a copy for each targeted output port for every received multicast packet. By targeted output port we refer to the output port that needs to receive the corresponding multicast packet. However, this can significantly increase the amount of traffic that needs to be forwarded from the input ports. For example, in the worst case where all packets are multicast and each multicast packet needs to be forwarded to all output ports, the increase in traffic at the input port would be N times - instead of 1 packet, N packets should be forwarded for each received multicast packet.

Workaround is to create, for each multicast flow, a multicast tree of output ports that belong to that flow, while the tree's root node is the input port that receives the multicast packets of that flow [13]. Multicast packets travel down the corresponding tree to all tree's nodes, where at each internal node, copies of the multicast packet are created for its children nodes. In this way, switch still operates as a unicast switch, but the number of additional copies created at each port is limited, thus, the problem of the input port overload with additional packet copies is limited.

The other way to adjust IQ based schemes to multicast traffic is to use a multicast switch fabric that enables simultaneous packet transmission to multiple output ports from one input port [3], [14]. Here, the additional packet copies are not created at the input ports and as a result, oversubscription of the input ports is avoided. Hence, packets are duplicated at the switching elements inside the switch fabric, and multicast controller is needed to efficiently configure the switch fabric and schedule packets. It is shown that the IQ multicast switch based on multicast capable switch fabric has better performance than the IQ multicast switch based on unicast fabric [15]. In [2], there is an exchange of control information between input and output ports in order to calculate multicast switch configuration to achieve efficient transfer of packets from input ports to output ports. In [16], multicast switch configuration is calculated in three phases: 1) Scheduling of multicast packets, 2) Scheduling of unicast packets, 3) Packet sending. Order of phases 1) and 2) is reversed between neighbouring calculation cycles to achieve a fair service of multicast and unicast traffic. The downside of the approach based on multicast switch fabric is the larger complexity of the multicast switch fabric compared to unicast switch fabric.

It is important to notice that the major downside of all IQ based solutions is a random pattern of switch configurations which limits the scalability in terms of supported switch throughput because the longest duration of calculation of one switch configuration determines the supported throughput. Even when the calculation operations are not complex, they still consume some time to be calculated, and this might represent a limiting factor when link speeds are 100Gbps and beyond.

In order to avoid the major downside of the IQ based solutions, load balanced Birkhoff-von Neumann (LB-BvN) architecture can be used [17]. LB-BvN uses two switch stages. The pattern of switch configurations is deterministic in both stages, thus, avoiding the major downside of the IQ based solutions since no switch configuration calculations are performed at all. The task of the first stage is to uniformly distribute incoming traffic over all input ports of the second stage. Then, the second stage forwards packets to their corresponding outputs. Central buffers are placed between two switching stages to store packets before forwarding them to the output ports. However, since the packets of the same flow can travel via different paths through the two stage architecture, the packets can experience different delays and these delay differences lead to out-of-sequence problem [17].

Multiple solutions are proposed to cope with the out-of-sequence problem. The first approach is to use resequencing buffers at the output ports. EDF (Earliest Deadline First) [17] and BF (Byte Focal) [18] use the resequencing buffers to solve the out-of-sequence problem. The drawback of these solutions is that the resequencing buffers increase the overall hardware complexity. Typically, the resequencing buffers have O(N^2) complexity or higher [18].

The second approach is to use frame based solutions, such as: FFF (Full Frames First) [19], PF (Padded Frames) [20], CR (Contention and Reservation) [21]. Here, incoming packets are organized in frames which consist of N consecutive packets of the same flow. Frame is sent only when it is completed. Each packet from the frame is sent to different central buffer. Since packets are sent only within the frames, all packets from the same frame experience the same delay, thus, packets are received in sequence. However, frame-based switches have significantly larger packet delays, especially at light traffic loads, because it takes some time to complete a full frame. For this reason, in some frame-based solutions, frames can be completed with dummy packets that are dropped at the output ports in order to reduce the overall packet delay [20]. But this leads to a problem of using the switch capacity to transfer dummy information.

The third solution that prevents out-of-sequence problem is FBSS (Feedback Based on Staggering Symmetry) [22], [23]. Thanks to the specific (staggering symmetry) connection patterns of both stages, the packet out-of-sequence problem is avoided. Staggering symmetry connection pattern means that if the central buffer i is connected to output port j, then, in the next iteration, the central buffer i is connected to input port j. Here, small central buffers, which can store up to N packets (one for each output port), advertise non-scheduled outputs. The central buffer i advertises non-scheduled outputs to output port j. Since input and output port j typically reside on the same line card, the output port j can pass the received information to the input port j. Input port j uses that information to schedule and send a packet to the central buffer i in the next iteration. Since each central buffer can have at most one packet for each of the outputs, all packets experience the same delay, thus, the packet out-of-sequence problem is avoided.

The fourth solution, LB-BvNGS (Load Balanced Birkhoff-von Neumann switch with Greedy Scheduling) [24] combines the best properties of IQ switches and LB-BvN switches. LB-BvNGS uses simple distributed greedy scheduling to schedule packets at the input ports for forwarding to the output ports. However, the scheduler is only responsible for packet selection at the input ports, it does not configure switch fabric, because configuration pattern of the switch fabric is deterministic (property of the LB-BvN class of switches). Furthermore, thanks to synchronized and same configuration pattern of the first and the second switching stage, a folded switch architecture is used where only one switch fabric is used for both switching stages unlike in most of the other LB-BvN based solutions where two physical switch fabrics must exist. LB-BvNGS originally works without packet out of sequence problem. LB-BvNGS can be modified to achieve even greater performance, but at the cost of using the resequencing buffers at the output ports to solve the packet out of sequence problem [24]. However, the size of resequencing buffers is limited to size of N, which is significantly lower than in the other LB-BvN solutions that use the resequencing buffers. As shown in [24], the FBSS and LB-BvNGS exhibit the best performance among the LB-BvN based solutions for the unicast traffic.

Most of the LB-BvN solutions are not dealing with the multicast support, but with the out-of-sequence problem of the unicast solutions. However, a deterministic switch configuration property provides great scalability for LB-BvN based switches. Therefore, LB-BvN architecture should be investigated for multicast support. Multicast support in FBSS is provided by adding the non-overlapping multicast queues at the input ports where each queue corresponds to one subset of the output ports [25]. When a multicast packet arrives to the switch, it is stored in multicast queues which correspond to destination output ports of that packet. In the scheduling process, packets from multicast queues have priority over unicast packets. At each input port, scheduler selects, from all multicast queues at that input port, a HOL (Head of Line) multicast packet which has the largest overlap with the set of non-scheduled outputs signaled by the central buffer as explained previously for the unicast FBSS solution. Afterwards, at the input port, destination output ports of that scheduled packet are updated to exclude the output ports that packet is sent to. Only when the packet is finally sent to all destination output ports, the packet is removed from the corresponding multicast queue at the input port. If none of the multicast packets can be sent, unicast packet is selected for some of the free outputs signaled by the central buffer. The number of multicast queues at the input ports needs to be carefully selected. If there is only one multicast queue (such queue corresponds to all output ports), then HOL packet is not removed from the queue until it is sent to all its destination ports. Thus, HOL blocking problem, especially under heavy load, can cause significant performance degradation. On the other side, if there are N multicast queues (each queue now corresponds to one output port), then packet replication can lead to input port overload with the created packet copies. Hence, the number of additional multicast queues is a trade-off between avoiding HOL blocking problem and creation of large number of packet copies.

III. MULTICAST LOAD BALANCED BIRKHOFF-VON NEUMANN SWITCH WITH GREEDY SCHEDULING

In order to explain the proposed multicast switch architecture, we first provide a brief description of the unicast load balanced Birkhoff-von Neumann switch with greedy scheduling (LB-BvNGS) because our proposed multicast architecture is based on the unicast LB-BvNGS architecture. Then, we explain the multicast controller that is used in combination with the unicast LB-BvNGS to achieve multicast switch architecture MLB-BvNGS (Multicast LB-BvNGS). In this paper, we assume fixed size packets. In the case of variable size packets, packet segmentation is performed at the input ports.

Let us consider a LB-BvNGS switch of size $N \times N$, where N is the number of input/output ports. In LB-BvNGS, time is divided in cycles that last N slots, where the slot represents duration of one fixed-size packet. LB-BvNGS has two switching stages, where deterministic connection pattern of both stages is the same: $j = (i + t)\mathbf{mod}N$, where i and j represent connected input and output ports of stage in slot t. Typically, input port k, output port k and central buffer k are implemented on the same line card. This property, along with the same connection patterns of both stages, enables the usage of folded architecture where only one physical switch is used instead of two. It is shown that folded physical switch in LB-BvNGS does not need speedup of switch configurations [24]. This property enables good scalability of LB-BvNGS since switch configuration speedup is more critical than the internal links speedup.

At each input port there are N queues. Packets which are destined to the same output port are stored in the same queue. During each cycle, input ports use greedy scheduling algorithm to choose packets that will be sent in the next cycle. In each cycle, input ports exchange N vectors Vc ($c = 0 \ldots N-1$), where each vector Vc corresponds to one central buffer c. Each bit in Vc corresponds to one output that can be reached via central buffer c. Therefore, each vector Vc comprises N bits. If the bit on the location j of the vector

Vc is set to 1, it means that central buffer c can accept packet destined to the output port j i.e. none of the input ports scheduled packet for that location in central buffer c. Otherwise, if the bit on the location j of the vector Vc is set to 0, then central buffer c already has a scheduled packet for output port j i.e. one of the input ports already scheduled packet for that location in central buffer c. When input port receives Vc, it determines non-occupied outputs and selects packet for one of them according to the longest queue policy. The selected output is marked as occupied in Vc and updated Vc is passed to the next input port in chain. For example, when input port i processes Vc, it forwards the vector Vc to the input port $(i-1)\mathbf{mod}N$. At the beginning of the cycle, each input port c processes vector Vc. Thus, in every slot, each input port processes only one vector, and vectors are sequentially exchanged between the inputs. Vc value at the beginning of the cycle shows that all outputs are unoccupied at the central buffer c. This property of processing only one vector during the slot makes LB-BvNGS very scalable.

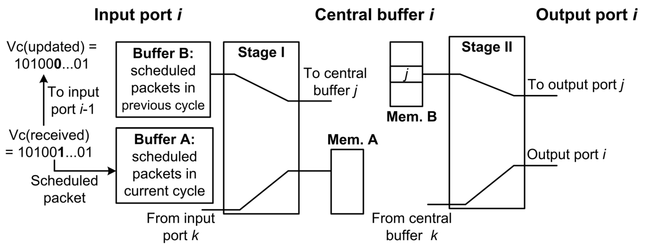

FIGURE 1. LB-BvNGS port architecture.

The scheduled packets are stored in the corresponding buffer for scheduled packets at the input port. Each input port comprises two such buffers. One buffer (Buffer A in Fig. 1) stores the packets scheduled in the current cycle, and the other buffer (Buffer B in Fig. 1) stores the packets scheduled in the previous cycle (these packets are forwarded to central buffers in the current cycle). The role of these two buffers is switched between the cycles. The size of each buffer is only N packets, since at most N packets can be sent from the buffer (Buffer B) or scheduled (Buffer A) in one cycle. These two buffers for scheduled packets at the input ports are needed to avoid packet out-of-sequence problem as explained in [24]. The packets scheduled in the current cycle are transmitted to central buffers in the next cycle.

The explained greedy scheduling ensures that each central buffer receives at most one packet for each of the outputs, so very small memories are used at the central buffers. Each central buffer contains only two RAM memories with N locations. In both memories, each location corresponds to one output port. During the cycle, one memory (Mem. A in Fig. 1) receives packets from the input ports, and those packets will be sent to the corresponding output ports in the next cycle. Other memory (Mem. B in Fig. 1) stores packets that arrived in the previous cycle, and these packets are sent to the output ports in the current cycle. The role of these two memories is switched between the cycles. It is shown in [24], that overall LB-BvNGS complexity is very low, while achieving excellent performance.

Fig. 1 shows the previously described architecture of port i of the LB-BvNGS switch. Switching stages are shown logically (separated), but physically they are the same switch fabric. During one slot, input port i is connected to central buffer j, and central buffer i is connected to output port j. Also, input k and central buffer k are connected to central buffer i and output port i, respectively. Input port i receives Vc from input port $(i+1)\mathbf{mod}N$ and selects the longest queue among the queues that correspond to outputs that are marked as free in Vc. The corresponding bit in Vc is flipped and Vc is passed to the input port $(i\text{-}1)\mathbf{mod}N$. The scheduled packet is placed in the corresponding buffer (Buffer A in Fig. 1) that contains packets scheduled for transmission in the next cycle. The packet scheduled in the previous cycle (stored in Buffer B in Fig. 1) for central buffer j is transmitted. Transmitted packet from input port k is written to memory A in central buffer i at the location corresponding to output that represents packet's destination. Packet from location j in memory B at central buffer i is transmitted to output port j. Roles of memories A and B are switched in the next cycle.

In order to add a multicast support to the previously described unicast LB-BvNGS architecture, we implement multicast controller. Multicast controller is responsible to create a binary multicast tree for each multicast flow. First, we explain the structure of one multicast binary tree as well as the packet forwarding in the tree. Then, we explain the construction and update of a binary multicast tree.

One binary multicast tree comprises root node and regular nodes. Root node is the root of the binary tree and it is placed at the input port where the packets of the corresponding multicast flow are received. Regular nodes are the output ports to which the packets of the corresponding multicast flow need to be forwarded. An example of one multicast tree for one multicast flow F is shown in Fig. 2.

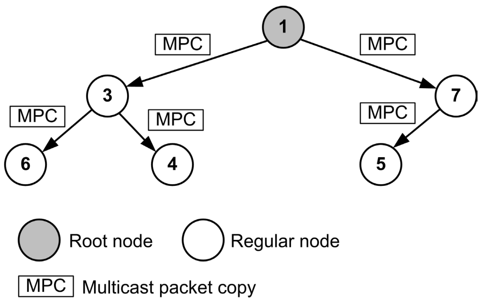

FIGURE 2. Binary multicast tree forwarding.

Node indices in Fig. 2 correspond to port indices. The input port 1 receives the multicast packets of flow F. Note that input port uses the multicast address from packet's header to perform the lookup to multicast forwarding table at the input port in order to determine to which flow the received multicast packet belongs. The multicast address can be multicast IP address, MPLS label, etc. For each received multicast packet of flow F, the input port 1 creates two copies of the packet, one for the output port 3 and one for the output port 7. These two copies are placed in the corresponding queues for the output ports 3 and 7, respectively. Thus, the copies of the multicast packets are forwarded using the unicast LB-BvNGS architecture. At the input port 1, internal multicast label f is added to each copy. Since, multicast packets are forwarded using the unicast LB-BvNGS architecture, there is a need to distinguish multicast packets from the unicast packets, and also to distinguish to which multicast flow does the multicast packet belong. Internal multicast label enables the ports to choose the correct multicast tree for the packet forwarding. Internal multicast label is retrieved from the multicast forwarding table as well as the IDs of the left and right child nodes. In the case when some child node does not exist, NULL value is retrieved indicating that the corresponding child does not exist. The described process of the multicast packet forwarding at the root node from the given example is shown in Fig. 3 that represents the system architecture of the root node multicast packet forwarding. Note that all modules in Fig. 3 are part of the MLB-BvNGS and only the last module in Fig. 3 (Queues for LB-BvNGS) is part of the original LB-BvNGS as this is the place where unicast and multicast packet flows are joint together into LB-BvNGS unicast architecture.

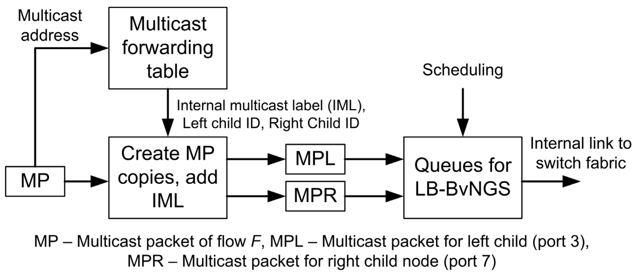
Multicast address
Multicast forwarding table
Internal multicast label (IML), Left child ID, Right Child ID
Scheduling
MP
Create MP copies, add IML
MPL
MPR
Queues for LB-BvNGS
Internal link to switch fabric
MP – Multicast packet of flow F, MPL – Multicast packet for left child (port 3), MPR – Multicast packet for right child node (port 7)

FIGURE 3. Root node multicast packet forwarding.

We also give the pseudocode of a root node packet forwarding. The pseudocode covers forwarding up to the moment where the packets are written to the corresponding LB-BvNGS queues i.e. where the unicast and multicast packet flows are joint into the LB-BvNGS unicast architecture that represents the basis of the MLB-BvNGS. Lines 11-13 of the pseudocode represent the part of the unicast LB-BvNGS architecture.

Multicast packet copy received at the output port 3 is detected to be a multicast packet. Thus, besides forwarding of the packet copy to the output, a lookup in internal multicast forwarding table is performed. Each output port contains internal multicast forwarding table, where for each internal multicast label there is a record of the child nodes in the corresponding binary multicast tree. If child nodes are detected, for each child, a copy of the packet is created and passed to the input port of the same index - in the given example it is the input port 3. Note that typically input and output port

Algorithm 1 Root Node Packet Forwarding Pseudocode

```
1: if received packet is multicast then
2:   multicast forwarding table lookup
3:   if left child exists then
4:     create left child copy and add IML
5:     write copy to corresponding LB-BvNGS queue
6:   end if
7:   if right child exists then
8:     create right child copy and add IML
9:     write copy to corresponding LB-BvNGS queue
10:  end if
11:else //unicast packet is received
12:  unicast forwarding table lookup
13:  write packet to corresponding LB-BvNGS queue
14: end if
```

reside on the same line card, thus, this packet passing from the output port to the input port of same index is low complexity operation. At the input port, these packet copies are added to the corresponding queues for the output ports. In the given example, at the output port 3, two packet copies are created for the output ports 4 and 6.

The described process of the multicast packet forwarding at the regular node 3 from the given example is shown in Fig. 4 that represents the system architecture of the regular node multicast packet forwarding. Note that all the modules in Fig. 4 are part of the MLB-BvNGS and only the Output queues and Queues for LB-BvNGS modules are part of the original LB-BvNGS.

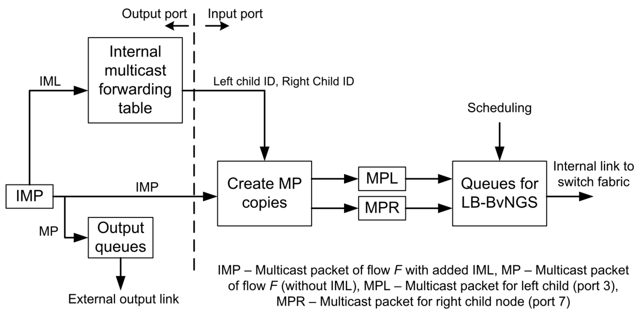
Output port
Input port
IML
Internal multicast forwarding table
Left child ID, Right Child ID
Scheduling
IMP
IMP
MP
Output queues
External output link
Create MP copies
MPL
MPR
Queues for LB-BvNGS
Internal link to switch fabric
IMP – Multicast packet of flow F with added IML, MP – Multicast packet of flow F (without IML), MPL – Multicast packet for left child (port 3), MPR – Multicast packet for right child node (port 7)

FIGURE 4. Regular node multicast packet forwarding.

We also give the pseudocode of regular node packet forwarding. Again, the forwarding is given up to the point where multicast copies are written to LB-BvNGS queues. Lines 12-13 of the pseudocode represent the part of the unicast LB-BvNGS architecture.

Only one packet copy (for output port 5) is created at the output port 7. In case when no child nodes are detected in the record retrieved from the internal multicast forwarding table, multicast packet reached the end of the binary multicast tree, thus, no additional packet copies are created. This happens at the output ports 4, 5 and 6 in the given example. The given example shows that using the binary multicast trees,

Algorithm 2 Regular Node Packet Forwarding Pseudocode

```
1: if received packet is multicast then
2:   write packet with removed IML to output queue
3:   internal multicast forwarding table lookup
4:   if left child exists then
5:     create left child copy
6:     write copy to corresponding LB-BvNGS queue
7:   end if
8:   if right child exists then
9:     create right child copy
10:    write copy to corresponding LB-BvNGS queue
11:  end if
12:else //unicast packet is received
13:    write packet to output queue
14:end if
```

the unicast LB-BvNGS architecture can be efficiently used for the multicast packet forwarding as well.

In order to create multicast forwarding trees in the network of routers, a protocols like PIM (Protocol Independent Multicast) are used [26]. Based on the work of these protocols, routers can determine for each multicast flow F, which input port of the router receives the multicast packets of the flow F, and to which output ports of the router the multicast packets of flow F need to be forwarded. The protocol messages are processed in the control plane. For that reason, we implement our multicast controller (MC) in the control plane. The multicast controller is responsible for creating and sending configuration messages to input and output ports of the router. In that way, our MC implementation can support software defined networking (SDN) paradigm where the control plane is decoupled from the actual router hardware and router in such case comprises only the forwarding plane. This represents additional reason why we decide to implement MC in the control plane. Based on multicast protocol (like PIM) messages, the MC creates and updates the binary multicast trees in the router. In order to create the binary multicast tree for some multicast flow F, there needs to be at least one output port to which the multicast packets of the multicast flow F need to be forwarded. Multicast tree is incrementally updated. When new output port needs to be added to the binary multicast tree, the node corresponding to that output port is added to the tree. Left first rule is applied, where the nodes in the next level are added by adding left branch child nodes first.

Fig. 5 shows the order of regular node additions to the binary multicast tree, where regular node indices given in Fig. 5 show the order in which the regular nodes are added to the tree. Note that indices given in Fig. 5 do not correspond to port indices, but they correspond to the order of additions to the tree. In this way, we achieve balanced tree with the minimal possible depth. Minimal possible depth is important to minimize the difference in delays of the copies of the same multicast packet because the output ports belonging

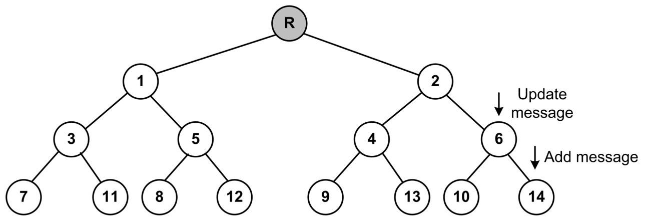

FIGURE 5. Binary multicast tree.

to the tree can have different number of hops from the root node. When a node is deleted from the tree, if the deleted node entry is not the last added node, the last added node is moved to the emptied position. By the last added node, we assume the node that has the largest index in the left first rule scheme. In this way, the tree always preserves the compact structure.

Fig. 5 and Fig. 6 show the configuration messages sent by MC in the most common multicast tree update scenarios.

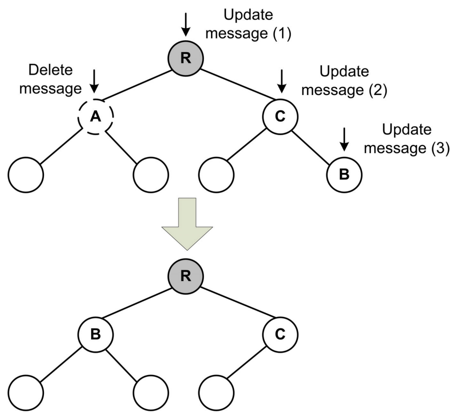

FIGURE 6. Regular node deletion.

The configuration messages sent to input ports (root nodes) are responsible for adding, updating and deleting the records in the multicast forwarding tables at the input ports. The record contains the internal multicast flow label and the IDs of the output ports that represent left and right child of the root node. Note that ID value NULL indicates that the corresponding child node does not exist.

The configuration messages sent to output ports (regular nodes) are responsible for adding, updating and deleting the records in the internal multicast forwarding tables at the output ports. The record contains the IDs of the output ports that represent left and right child in the corresponding multicast tree. Note that ID value NULL indicates that the corresponding child node does not exist. Fig. 5 shows the configuration messages sent from the MC in the case of the last added node (node 14). In this example, MC sends two configuration messages when a new node is added. One *update* message is necessary to update the corresponding record in the parent node with the ID of the added child node in the internal multicast forwarding table. Note that if the parent node is the root node, the updated record is in the multicast forwarding table. In the given example, the record in internal multicast forwarding table at the output port that corresponds to node at position 6 is updated with the right child ID that is set to value corresponding to the regular node (i.e. output port ID) that is added to the binary multicast tree (position 14 in the tree). The other configuration message sent by MC is *add* message that adds record to the internal multicast forwarding table of the regular node (i.e. corresponding output port) added to the binary multicast tree (at position 14 in the given example). Since the added node does not have any children, the added record contains NULL value for left and right child nodes.

Fig. 6 shows the most complicated case when a node is removed from the binary multicast tree and the removed node is not the last added node. In this most complicated case, MC needs to send four configuration messages. *Delete* message removes the corresponding record from the internal multicast forwarding table at the deleted regular node (node A in the given example). The last added node in the given example is the regular node B. Node B needs to be moved to the place of the deleted node (node A in the given example). This movement requires update in the node B to adjust to its new position in the tree. Also, updates are required in the parent node of the node B (regular node C in the given example) and in the parent node of the deleted node (root node R in the given example) in order to maintain the proper multicast packet forwarding through the binary multicast tree. These two parent nodes (nodes R and C) experience changes in the child nodes due to the node A removal and the node B position change. Thus, MC sends *update* message to the regular node B to update it with the child nodes of the removed regular node A. MC also sends *update* message to the root node R to update its left child with the node B (instead of the deleted node A) and *update* message to the regular node C to update its right child to NULL since the node B is moved to another position in the binary multicast tree.

As we can see from the given examples, MC is the one that keeps track of all the binary multicast trees and updates the structure of these trees based on the multicast protocol messages (like PIM messages) exchanged with the other routers. Based on the structure updates, MC creates configuration messages and sends them to the forwarding plane of the router in order to properly update the multicast forwarding tables at the affected router ports.

Unlike the most of the other existing multicast solutions, we derive the minimum internal link speedup required for the non-blocking property of the proposed MLB-BvNGS for any admissible traffic scenario. Also, even without the speedup, the proposed MLB-BvNGS achieves great performance for the typical traffic scenarios that are analyzed in the literature. The proposed MLB-BvNGS is non-blocking for any admissible traffic scenario if the speedup of five is used. In order to prove this statement, we introduce the following assumptions. The size of the switch is $N \times N$, where N is the number of input/output ports. The packets have a fixed size that is equal to duration of one slot. Now, let us observe some admissible traffic scenario. Since the traffic is admissible, there is $m \in Z^+$ such that the traffic in the interval $m \cdot N$ is admissible i.e. there is a period that lasts integer number of cycles where the traffic is admissible. Note that Z^+ is the set of positive integers ($\geq$ 1). There are at most $m \cdot N$ packets for each of the outputs in the observed interval (that lasts $m \cdot N$ slots) because the traffic is admissible. Given the input link capacity limits, at each input port at most $m \cdot N$ incoming packets can be received in the observed interval. Now, we observe some input port i and the packet that should be forwarded from that input port to some output port j. The worst case in forwarding some packet from the input port i is the following: 1) all the incoming packets received at the input port i are multicast packets with at least two destination ports; 2) all the packets for the output port i are the multicast packets and for these multicast packets port i is the regular node in the multicast tree that is not leaf. In the worst case, the considered packet must wait forwarding of all the packets from the input port i that includes the copies of the multicast packets received at the input port i as the root node and as the regular node. There are $2 \cdot m \cdot N$ packets where the input port i plays the role of the root node. This is the consequence of point 1) in the analyzed worst case: each packet is multicast with at least two output port destinations, thus, there are $2 \cdot m \cdot N$ copies. There are $m \cdot N$ packets where the input port i plays the role of the regular non-leaf node in the multicast tree. This is the consequence of point 2) in the analyzed worst case and the fact that the traffic is admissible: since the traffic is admissible there can be at most $m \cdot N$ copies for the destination output port i. Since the port i is non-leaf node, in the worst case $2 \cdot m \cdot N$copies need to be forwarded (both child nodes exist). Also, there are at most $m \cdot N$ packets destined for the output port j since the traffic is admissible. In the worst case, the considered packet needs to wait forwarding of at most $4 \cdot m \cdot N - 1$ packets from the input port i and forwarding of $m \cdot N - 1$ packets for the output port j from the other input ports before the considered packet can be forwarded. This gives a total of $5 \cdot m \cdot N - 2$ packets, thus speedup of five is required to forward all the traffic for the period of $m \cdot N$ slots.

The goal of a speedup is to enable faster forwarding of packets from input ports to output ports in order to avoid potential congestions at the input ports [27]. Also, the speedup decreases packet delays [27]. The larger speedup value as a consequence has a higher implementation cost [28]. Typically, in the packet switches that use a speedup, speedup is required for both, the internal links and the switch fabric [28]. If the speedup value is *SP*, then the internal links rate is *SP* times higher than the external links rate. The straightforward way to implement the internal link speedup is to simply use faster links inside the router. However, the maximal speedup value in this approach is limited by the external link rate value and the used technology. In the case where this straightforward approach is not able to implement a desired *SP* value, then the desired *SP* can be achieved by implementing each internal link as an aggregate of several links to achieve the desired internal link rate. This expansion of the straightforward approach additionally increases the implementation costs since the switch fabric should accommodate (at its inputs and outputs) bundles of links instead of individual links. The speedup of switch fabric requires faster reconfigurations of the switch fabric in order to properly switch the packets that are incoming over internal links with speedup [28]. Usually, the switch fabric speedup is considered to be more critical from the design point of view [28]. However, the LB-BvNGS does not require speedup of switch fabric even in the case when the internal links use speedup [24]. Since, the MLB-BvNGS relies on the LB-BvNGS for packet forwarding, the same property holds for the MLB-BvNGS as well. Because of this, we believe that the MLB-BvNGS implementation cost increase if the speedup of five is used is acceptable since only the internal links use speedup. However, we emphasize also that the MLB-BvNGS exhibits great performance even when no internal links speedup is used as shown in section IV, so the lower value of speedup can be used.

IV. PERFORMANCE ANALYSIS

In order to analyze the performance of the proposed multicast LB-BvNGS (MLB-BvNGS) we develop the simulation model for the MLB-BvNGS. Our goal is to test the MLB-BvNGS switching performance (performance in the data plane) and compare it to the other existing multicast switch solutions. For this reason, the simulation model does not differ the location of the control plane i.e. whether the control plane is decoupled from the data plane or not. The simulation model is written in C language.

MLB-BvNGS simulation model pseudocode gives the overview of the MLB-BvNGS simulation process. Note that we omit the simulation of speedup from the pseudocode because we do not use speedup in the performance analysis presented in this section. The simulation starts with the initialization of the following parameters: simulation duration in number of slots (T_{sim}), switch dimension - number of input/output ports (N), traffic scenario, traffic generator parameters. Traffic generator parameters depend on the selected traffic scenario. More details on the traffic scenarios and the traffic generator parameters are given later in this section.

After the initialization, the simulation process enters the while loop where each loop iteration represents one slot - time in simulation is measured in slots. We set T_{sim} to 1 million slots. In case when slot corresponds to beginning of the new cycle, all Vc vectors are initialized to all 1s (all central buffers are marked as completely free as explained in Section III). Then, for loop passes through each input port and the following functions are performed at each input port:

1) Incoming traffic is generated according to the selected traffic scenario. The generated packets are written to corresponding input queues at the input ports. There are N input queues at each input port, and each input queue corresponds to one output port. The input queues are modeled as lists, where each element of the list represents one packet.

Algorithm 3 MLB-BvNGS Simulation Model Pseudocode

```
1: simulation parameter initialization
2: $t = 0$
3: while ($t < T_{sim}$) do
4:   if ($t$ mod $N = 0$) then
5:     init all $Vc$ vectors to free
6:   end if
7:   for each input do
8:     generate incoming traffic and write to corresponding input queues
9:     select packet for the next cycle and update $Vc$
10:    send scheduled packet to central buffer
11:  end for
12:  for each central port do
13:    send packet to output port
14:  end for
15:  for each output port do
16:    send packet to output external link
17:    if multicast packet and at least one child exists then
18:      write copies to corresponding input queues
19:    end if
20:  end for
21:  for each input do
22:    send $Vc$ to neighbour input port
23:  end for
24:  $t++$
25: end while
```

List element (packet) comprises the following information: timing information for each point in the path of the packet from the input port to the output port (time when the packet entered the input port from the traffic generator, time when the packet is sent to the central buffer, time when the packet is sent to the output port, time when the packet re-entered the input port - this last time is relevant only for the multicast packet copies created at the output port), type of the packet (unicast/multicast), input port id, output port id (used only for the unicast packets), subtree (used only for the multicast packets). Subtree represents the remaining part of the multicast tree that the copies of the corresponding multicast packet need to traverse. Subtree is embedded in the packet to avoid simulating the process of the internal multicast forwarding table lookup because simulating the lookup would increase the complexity of the simulator without adding the value to the performance analysis of the packet switching. The timing information is used not only for monitoring and analysis purposes, but also for preventing the packets to traverse multiple stages in one slot which would violate the simulation correctness. For example, if the packet is added to the input queue at the current slot, it can not be scheduled in the current slot - the timing information says that packet entered the input queue in the current slot, thus it is not considered for the scheduling.

2) Packet scheduling for the next cycle. By inspecting the vector Vc and non-empty input queues, the longest queue

among the input queues that are allowed by Vc to schedule the packet is selected. The scheduled packet is written to the scheduled packets buffer (Buffer A in Fig. 1) that is also modeled as a list. Vc is updated by marking the central buffer that is selected by the scheduler as occupied.

3) Sending the packet scheduled in the previous cycle to the corresponding central buffer. Corresponding central buffer is the central buffer to which the input port is currently connected according to $j = (i + t)\mathbf{mod}N$ formula, where i is the id of the input port and j is the id of the central buffer. The packet from the list that corresponds to Buffer B in Fig. 1 is sent to corresponding central buffer. Note that the lists that correspond to Buffers A and B switch their roles between the cycles as explained in Section III. The sent packet is written to the list that correspond to Memory A in Fig. 1 at the corresponding central buffer.

After the input port events simulation, for loop that passes through each central buffer is performed. The packet from Memory B in Fig. 1 is sent to the corresponding output port. Corresponding output port is the output port to which the central buffer is currently connected according to $j = (i + t)\mathbf{mod}N$ formula, where i is the id of the central buffer and j is the id of the output port. Note that the lists that correspond to Memories A and B switch their roles between the cycles as explained in Section III. The packet is written to the list that corresponds to output buffer at the output port.

At the output port the packets are sent from the output queue to the external link and the total delay is measured for the packet. The total delay represents the time that has passed between the time when packet entered the input port from the traffic generator and the time when packet was sent to external output link. If the packet is multicast at the output buffer, besides sending the packet to the external output link, the subtree in the list element (packet) is inspected. If there is at least one child, the corresponding packet copies (one or two, depending whether one or two child nodes exist) are created and written to the corresponding input queues. The subtree in each packet copy is updated so that left child gets left part of the subtree and right child gets right part of the subtree, with the subtree root node removed (it is removed because the packet has reached that output port).

At the end of simulation of the events during one slot, each input port sends updated Vc to its neighbour input port. For the input port i, the neighbour input port is $(i - 1)\mathbf{mod}N$.

We compare the proposed multicast LB-BvNGS (MLB-BvNGS) to the recent IQ multicast schemes proposed in [2], [16]. We denote these two IQ schemes as M-IQ1 [2] and M-IQ2 [16]. We also compare MLB-BvNGS to the multicast scheme based on LB-BvN that is proposed in [25]. We denote this scheme as M-BvN. We test two M-BvN schemes that differ in the number of multicast queues because the number of multicast queues has significant impact on performance. Namely, we select schemes with 2 and 16 multicast queues that we denote as M-BvN2 and M-BvN16, respectively. We compare the schemes in terms of average packet delay D_{avg}. Speedup is not used in any of the schemes. We show the results for three admissible traffic scenarios: Bernoulli uniform mixing traffic (BUMT), Bernoulli uniform multicast traffic (BUMuT) and bursty mixing traffic (BMT) [25].

The traffic generator parameters for the BUMT scenario are: P - probability that packet arrives in slot, P_{m} - probability that the packet is multicast, F_{min} - minimal fanout of the multicast packets, F_{max} - maximal fanout of the multicast packets. The traffic generator works in the manner described by the following pseudocode. The given pseudocode is performed at each input port in every slot.

Algorithm 4 BUMT Traffic Generator Pseudocode

```
1: generate random number $x$ in range 0 to 1
2: **if** ($x < P$) **then**//packet arrives
3:    generate random number $y$ in range 0 to 1
4:    **if** ($y < P_m$) **then**//multicast packet arrives
5:      generate random integer $z$ in range $F_{\text{min}}$ to $F_{\text{max}}$
6:      randomly select $z$ outputs and create multicast tree
7:      create packets (list elements) and write to
corresponding input queues
8:    **else** //unicast packet arrives
9:      select random output $o$ from range $0$ to $N - 1$
10:    create packet (list element) and write to corresponding
input queue
11:   **end if**
12:**end if**
```

When the traffic generator creates a multicast packet, multicast tree associated to that packet is created. Given the randomly generated fanout z, outputs are randomly selected one by one and added to the multicast tree following the addition order shown in Fig. 5. In this way, we simulate the random creation of the multicast tree, where the outputs join the multicast flow in random order and consequently they are added to the multicast tree following that joining order. The generated multicast packets have a corresponding subtree embedded in the list element that represents the packet. Left child holds the left subtree of the generated multicast tree, and right child holds the right subtree of the generated multicast tree.

BUMuT scenario is very similar to BUMT scenario, with only one difference - all arriving packets are multicast i.e. P_{m} is set to 1.

Fig. 7, Fig.8 and Fig. 9 show the average packet delay D_{avg} for BUMT scenario where P_{m} is set to 0.25, 0.5 and 0.75, respectively. The switch size is set to 32 input/output ports. F_{min} is set to 2 and F_{max} is set to 32. Fig. 10 shows the average packet delay D_{avg} for BUMuT scenario that practically represents special case of BUMT where P_{m} is set to 1. We show the D_{avg} for high and very high loads, because significant difference between schemes begins to show at loads higher than 0.9.

All schemes exhibit stable behaviour in BUMT and BUMuT scenarios, and MLB-BvNGS achieves the lowest D_{avg} at the highest loads in all four cases shown

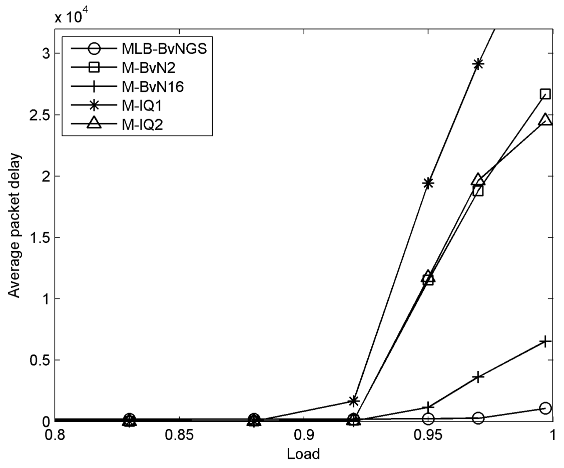

FIGURE 7. Average packet delay for BUMT scenario for $P_m = 0.25$.

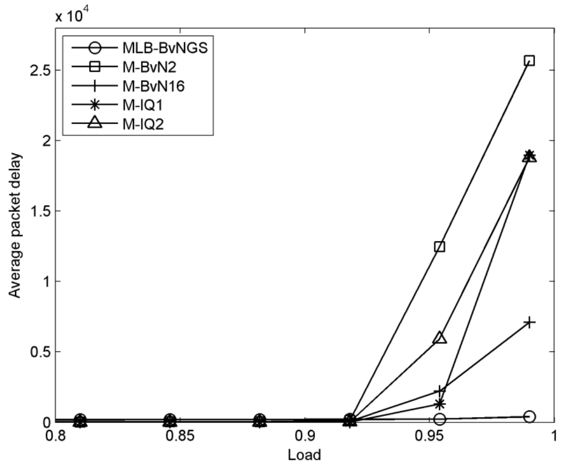

FIGURE 8. Average packet delay for BUMT scenario for $P_m = 0.5$.

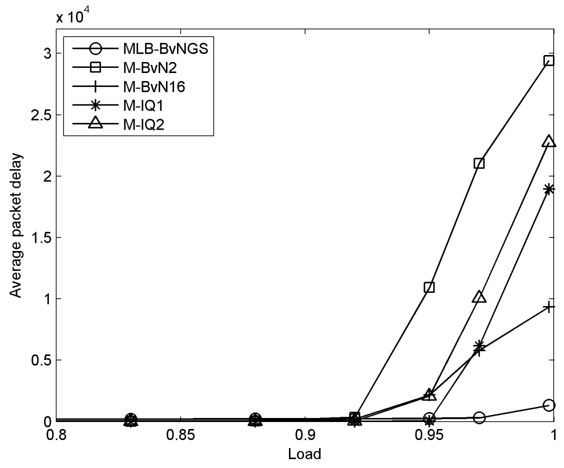

FIGURE 9. Average packet delay for BUMT scenario for $P_m = 0.75$.

in Fig. 7 - Fig. 10. Only in the case shown in Fig. 7 (BUMT $P_{\mathrm{m}} = 0.25$), the M-IQ1 exhibits poor performance at the highest loads, which suggests that M-IQ1 scheme is more

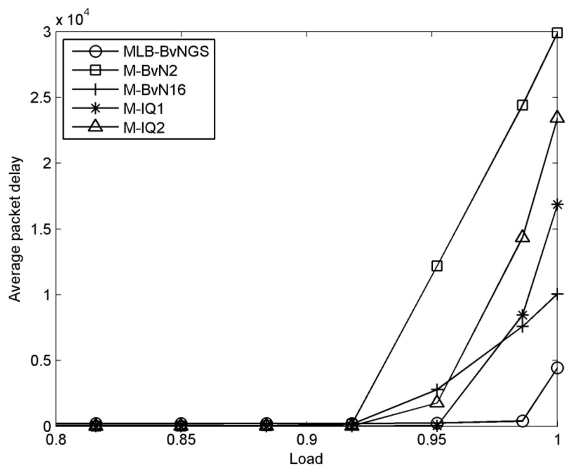

FIGURE 10. Average packet delay for BUMuT scenario.

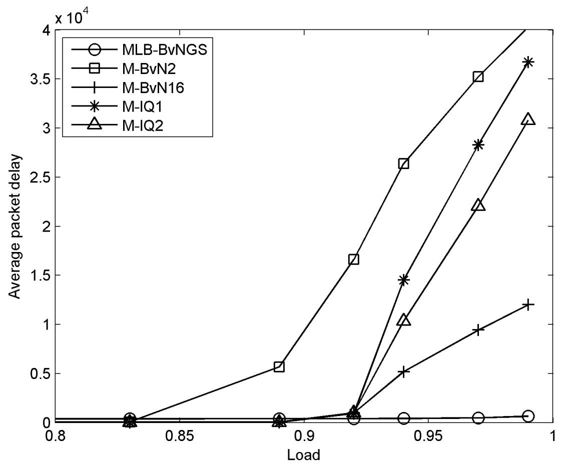

FIGURE 11. Average packet delay for BUMT scenario for $N = 64$.

adjusted to multicast traffic than to the unicast traffic in the case of BUMT scenario. M-BvN2 scheme has lower performance than M-BvN16 scheme, because there are less multicast queues, thus, HOL blocking has significant impact on performance, especially at high loads. For this reason, M-BvN2 scheme has the worst performance in all tested scenarios with the exception of the scenario shown in Fig.7. On the other hand, M-BvN16 scheme creates more packet copies and requires either faster memories to store all the created copies in one slot or multiple physical memory instances. But, since there are more multicast queues at the input port, the effects of the HOL blocking are significantly decreased in the case of M-BvN16 scheme. For this reason, the M-BvN16 scheme achieves the second best performance at the highest loads in the tested scenarios.

In order to inspect the schemes scalability and behavior in the case of greater switch sizes, we show the average packet delay D_{avg} for BUMT scenario where P_{m} is set to 0.5 and for the BUMuT scenario, when the switch size N is set to larger values - 64 and 128. All other traffic scenario parameters are the same as in traffic scenarios given for the switch size 32. Fig. 11 and Fig. 12 show the results for the switch size

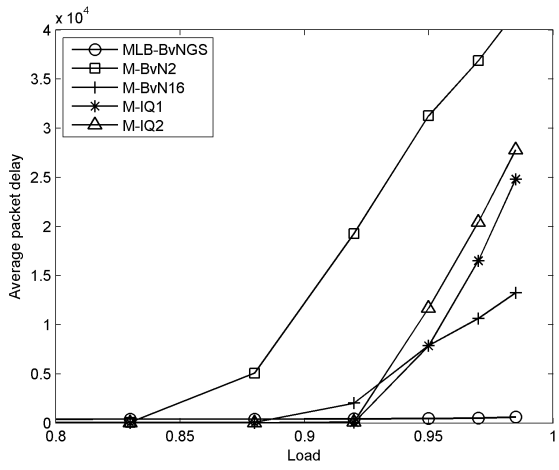

FIGURE 12. Average packet delay for BUMuT scenario for $N = 64$.

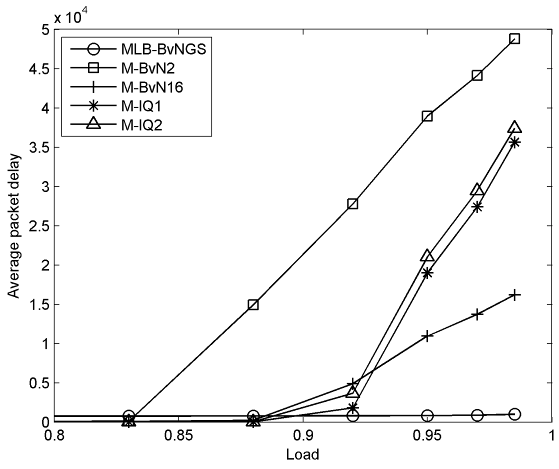

FIGURE 14. Average packet delay for BUMuT scenario for $N = 128$.

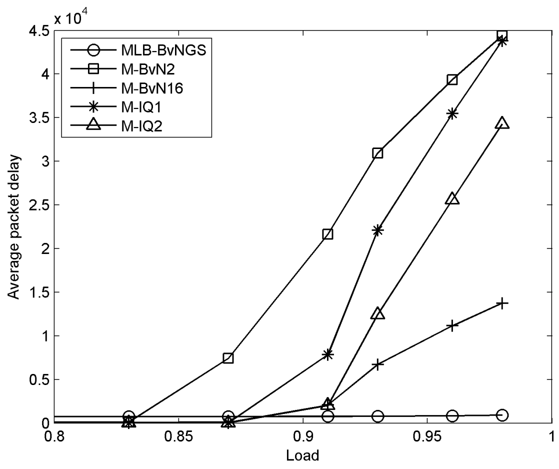

FIGURE 13. Average packet delay for BUMT scenario for $N = 128$.

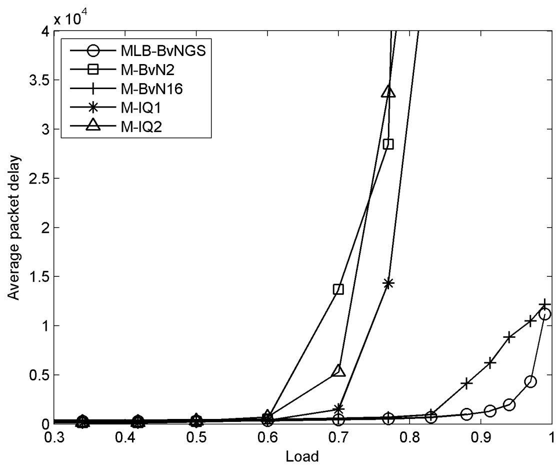

FIGURE 15. Average packet delay for BMT scenario for $P_m = 0.25$.

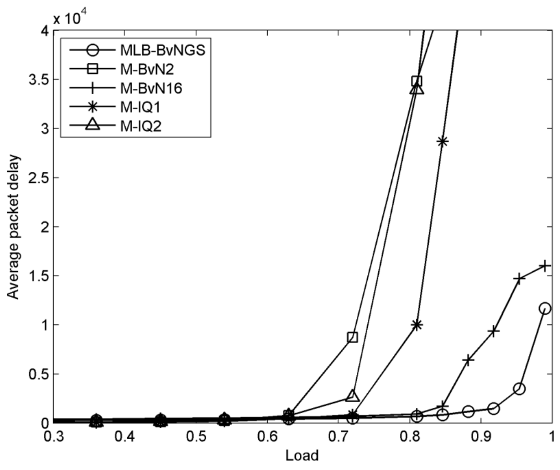

FIGURE 16. Average packet delay for BMT scenario for $P_m = 0.5$.

64 for BUMT and BUMuT scenarios, respectively. Fig. 13 and Fig. 14 show the results for the switch size 128 for BUMT and BUMuT scenarios, respectively. The relation between the schemes is similar to relations between the schemes in the tested BUMT and BUMuT scenarios for the switch size 32. Again, the M-BvN2 scheme exhibits the worst performance because of the HOL blocking problem, while the M-BvN16 exhibits the second best performance thanks to larger number of multicast queues that minimize the HOL negative effect. MLB-BvNGS still achieves the best performance indicating good scalability. M-IQ1 and M-IQ2 exhibit similar behavior like in tested scenarios for $N = 32$. M-IQ1 achieves better performance for the BUMuT scenario, while the M-IQ2 achieves better performance for the BUMT scenario, which again suggests that M-IQ1 is more suitable for cases when multicast traffic has more share in the overall traffic. Probable reason for this behaviour is that the M-IQ2 services unicast and multicast flows fairly, unlike the M-IQ1.

BMT traffic scenario uses ON/OFF model to simulate bursty traffic [25]. There are ON and OFF states for each input port. Packets arrive only in ON state. The traffic generator parameters for the BMT scenario are: P - probability of packet arrival, s - average burst size, P_m - probability that the burst is multicast, F_min - minimal fanout of the multicast

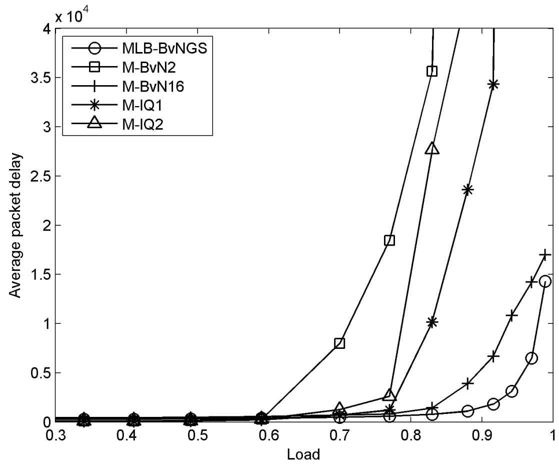

FIGURE 17. Average packet delay for BMT scenario for $P_m = 0.75$.

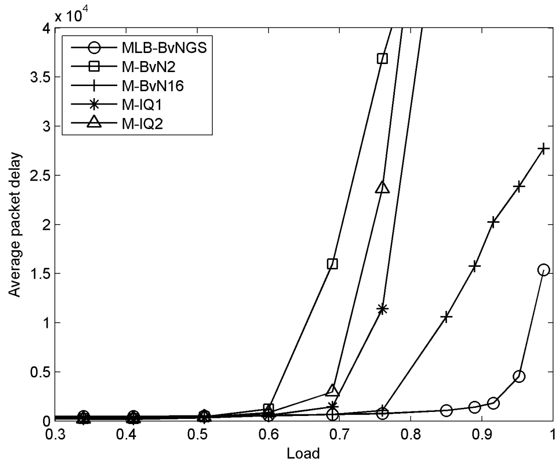

FIGURE 18. Average packet delay for BMT scenario for $N = 64$.

packets, $F_{\max}$ - maximal fanout of the multicast packets. The probabilities P_{on} and P_{off} are derived from the average burst size s and the probability of packet arrival P [25]. The probability to switch from OFF to ON state P_{on} is equal to $P/[s*(1-P)]$, while the probability to switch from ON to OFF state P_{off} is equal to $1/s$ [25]. Packets that belong to the same burst have the same destination ports. In the case of multicast packets, fan-out and destination ports are determined at the beginning of the burst in the same way as in BUMT traffic scenario. The traffic generator works in the manner described by the following pseudocode. The given pseudocode is performed at each input port in every slot. Note that burst element represents the pattern for packets in the burst. In the case of unicast burst, the burst element carries info about the output port to which the packets from the burst are destined for. The same applies for the multicast burst where the info about the output destinations is saved as the multicast tree.

Fig. 15, Fig. 16 and Fig. 17 show the average packet delay D_{avg} for BMT scenario where P_{m} is set to 0.25, 0.5 and 0.75, respectively. The switch size is set to 32 input/output ports. $F_{\min}$ is set to 2 and $F_{\max}$ is set to 32. Average burst

Algorithm 5 BMT Traffic Generator Pseudocode

```
1: if ($state = OFF$) then
2:   generate random number $x$ in range 0 to 1
3:   if ($x < P_{on}$) then//switch to ON state
4:     state = ON
5:     generate random number $y$ in range 0 to 1
6:     if ($y < P_m$) then//multicast burst
7:       generate random integer $z$ in range $F_{\min}$ to $F_{\max}$
8:       randomly select $z$ outputs and create multicast tree
9:       create burst element
10:    else //unicast burst
11:      select random output $o$ from range 0 to $N$-1
12:      create burst element
13:    end if
14:  end if
15:else//input port is in ON state
16:   create packets out of burst element and write to input queues //1 unicast packet or 2 multicast packets
17:  generate random number $w$ in range 0 to 1
18:  if ($x < P_{off}$) then//switch to OFF state
19:    state = OFF
20:  end if
21:end if
```

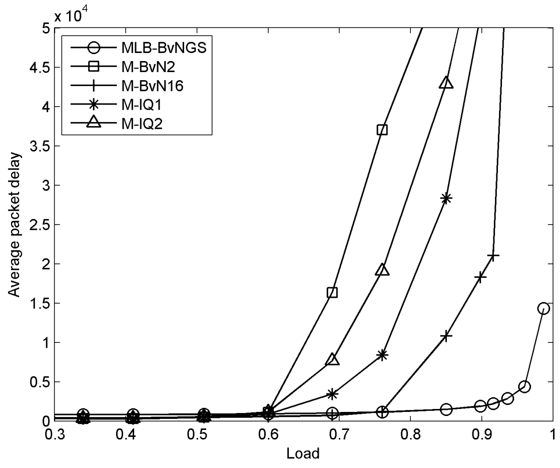

FIGURE 19. Average packet delay for BMT scenario for $N = 128$.

size is set to $s = 30$ as in [25]. BMT scenario puts more burden to switch architecture due to the burstiness of the traffic. As expected, all compared architectures exhibit higher D_{avg} than in the BUMT and BUMuT scenarios. Also, some of the schemes start to exhibit unstable behavior. M-IQ2 and M-BvN2 schemes are the first to become unstable. M-IQ1 scheme is slightly better than M-IQ2 and M-BvN2 as it becomes unstable at higher load values than the M-IQ2 and M-BvN2 schemes. MLB-BvNGS and M-BvN16 are stable even at very high loads, and again MLB-BvNGS achieves the lowest D_{avg} at high loads.

Fig. 18 and Fig. 19 show the average packet delay D_{avg} for BMT scenario when the switch size is increased to 64 and 128, respectively. The relation between the schemes is the same as in the BMT scenarios for $N = 32$. However, the M-BvN16 scheme has a decrease in performance at the highest loads as the switch size grows, suggesting that HOL blocking effect increases as the switch size increases. For this reason, the M-BvN scheme would have to use larger number of multicast queues for larger switch size but that would increase the implementation costs of the M-BvN scheme.

V. CONCLUSION

In this paper, we propose a novel multicast scheme based on the LB-BvN architecture. Performance comparison shows that LB-BvN based solutions are more stable at high loads than IQ based solutions. Also, our proposed MLB-BvNGS outperforms the other LB-BvN based multicast solution at very high loads. The tests show that MLB-BvNGS exhibits great stability even under very high loads. This good performance in combination with low complexity architecture presents MLB-BvNGS as very promising multicast solution.

REFERENCES

[1] H. N. Saha, A. Mandal, and A. Sinha, "Recent trends in the Internet of Things," in *Proc. IEEE 7th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Las Vegas, NV, USA, Jan. 2017, pp. 1–4.

[2] J. Xiao and K. L. Yeung, "Iterative multicast scheduling algorithm for input-queued switch with variable packet size," in *Proc. IEEE 30th Can. Conf. Electr. Comput. Eng. (CCECE)*, Windsor, ON, Canada, Apr. 2017, pp. 1–4.

[3] P. Giaccone, M. Pretti, D. Syrivelis, I. Koutsopoulos, and L. Tassiulas, "Design and implementation of a belief-propagation scheduler for multicast traffic in input-queued switches," *Comput. Commun.*, vol. 103, pp. 141–152, May 2017.

[4] L. Vu, V. L. Cao, Q. U. Nguyen, D. N. Nguyen, D. T. Hoang, and E. Dutkiewicz, "Learning latent distribution for distinguishing network traffic in intrusion detection system," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Shanghai, China, May 2019, pp. 1–6.

[5] F. Coras, J. Domingo-Pascual, F. Maino, D. Farinacci, and A. Cabellos-Aparicio, "Lcast: Software-defined inter-domain multicast," *Comput. Netw.*, vol. 59, pp. 153–170, Feb. 2014.

[6] R. Canonico and S. P. Romano, "Leveraging SDN to improve the performance of multicast-enabled IPTV distribution systems," *IEEE Commun. Standards Mag.*, vol. 1, no. 4, pp. 42–47, Dec. 2017.

[7] M. Shahbaz, L. Suresh, J. Rexford, N. Feamster, O. Rottenstreich, and M. Hira, "Elmo: Source routed multicast for public clouds," in *Proc. ACM Special Interest Group Data Commun.*, Beijing, China, Aug. 2019, pp. 458–471.

[8] C. Koch, S. Hacker, and D. Hausheer, "VoDCast: Efficient SDN-based multicast for video on demand," in *Proc. IEEE 18th Int. Symp. A World Wireless, Mobile Multimedia Netw. (WoWMoM)*, Macau, China, Jun. 2017, pp. 1–6.

[9] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Trans. Netw.*, vol. 7, no. 2, pp. 188–201, Apr. 1999.

[10] W. Zhu and M. Song, "Integration of unicast and multicast scheduling in input-queued packet switches," *Comput. Netw.*, vol. 50, no. 5, pp. 667–687, Apr. 2006.

[11] B. Hu, F. Fan, K. L. Yeung, and S. Jamin, "Highest rank first: A new class of single-iteration scheduling algorithms for input-queued switches," *IEEE Access*, vol. 6, pp. 11046–11062, Feb. 2018.

[12] M. Petrovic, A. Smiljanic, and M. Blagojevic, "Design of the switching controller for the high-capacity non-blocking Internet router," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 8, pp. 1157–1161, Aug. 2009.

[13] M. Blagojević and A. Smiljanić, "Design of multicast controller for high-capacity Internet router," *IET Electron. Lett.*, vol. 44, no. 3, pp. 255–256, Jan. 2008.

[14] M. A. Marsan, A. Bianco, P. Giaccone, E. Leonardi, and F. Neri, "Multicast traffic in input-queued switches: Optimal scheduling and maximum throughput," *IEEE/ACM Trans. Netw.*, vol. 11, no. 3, pp. 465–477, Jun. 2003.

[15] Z. Čiča, "Non-blocking frame based multicast scheduler for IQ switches," *Electron. Lett.*, vol. 52, no. 4, pp. 285–287, Feb. 2016.

[16] J. Xiao, K. L. Yeung, and S. Jamin, "Pipelined scheduler for unicast and multicast traffic in input-queued switches," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Washington, DC, USA, Dec. 2016, pp. 1–6.

[17] C.-S. Chang, D.-S. Lee, and C.-M. Lien, "Load balanced Birkhoff-von Neumann switches, part II: Multi-stage buffering," *Comput. Commun.*, vol. 25, no. 6, pp. 623–634, Apr. 2002.

[18] Y. Shen, S. Panwar, and H. Chao, "Design and performance analysis of a practical load-balanced switch," *IEEE Trans. Commun.*, vol. 57, no. 8, pp. 2420–2429, Aug. 2009.

[19] I. Keslassy and N. McKeown, "Maintaining packet order in two-stage switches," in *Proc. 21st Annu. Joint Conf. IEEE Comput. Commun. Societies*, New York, NY, USA, Jun. 2002, pp. 1032–1041.

[20] J. J. Jaramillo, F. Milan, and R. Srikant, "Padded frames: A novel algorithm for stable scheduling in load-balanced switches," *IEEE/ACM Trans. Netw.*, vol. 16, no. 5, pp. 1212–1225, Oct. 2008.

[21] C.-L. Yu, C.-S. Chang, and D.-S. Lee, "CR switch: A load-balanced switch with contention and reservation," *IEEE/ACM Trans. Netw.*, vol. 17, no. 5, pp. 1659–1671, Oct. 2009.

[22] B. Hu and K. L. Yeung, "Feedback-based scheduling for load-balanced two-stage switches," *IEEE/ACM Trans. Netw.*, vol. 18, no. 4, pp. 1077–1090, Aug. 2010.

[23] A. Huang and B. Hu, "The optimal joint sequence design in the feedback-based two-stage switch," in *Proc. IEEE ICC*, Sydney, NSW, Australia, Jun. 2014, pp. 3031–3036.

[24] S. Durkovic and Z. Cica, "Birkhoff-von Neumann switch based on greedy scheduling," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 13–16, Jan. 2018.

[25] B. Hu and K. L. Yeung, "Multicast scheduling in feedback-based two-stage switch," in *Proc. Int. Conf. High Perform. Switching Routing*, Paris, France, Jun. 2009, pp. 1–6.

[26] J. Ko, S. Park, and E. Lee, "An extended PIM-SM for efficient data transmission in IPTV services," in *Proc. 2nd IEEE Int. Conf. Netw. Infrastruct. Digit. Content*, Beijing, China, Sep. 2010, pp. 115–119.

[27] B. Prabhakar and N. McKeown, "On the speedup required for combined input and output queued switching," in *Proc. IEEE Int. Symp. Inf. Theory*, Cambridge, MA, USA, Aug. 1998, pp. 165–180.

[28] B. Hu, K. L. Yeung, and C. He, "On iterative scheduling for input-queued switches with a speedup of 2-1/N," in *Proc. IEEE HPSR*, Vancouver, BC, Canada, Jun. 2014, pp. 26–31.

SRDJAN DURKOVIC received the B.S. degree in telecommunications from the Faculty of Electrical Engineering, University of Montenegro, in 2013, and the M.S. degree in telecommunications from the School of Electrical Engineering, University of Belgrade, in 2014, where he is currently pursuing the Ph.D. degree. His research interests include packet switching, scheduling algorithms, and multicast.

ZORAN ČIČA (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in telecommunications from the School of Electrical Engineering, University of Belgrade, Serbia, in 2002, 2007, and 2012, respectively. In 2002, he joined the School of Electrical Engineering, University of Belgrade, where he is currently an Associate Professor. His research interests include packet switching, communication protocols, high-speed networks, and communication hardware design.