Fair Packet Dropping

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

***Abstract*—Many scheduling algorithms were proposed to provide fair service of flows passing through a buffer before the congested link. As we will show, fairness can be provided only if packets are dropped appropriately, while the fair scheduling algorithms is applied. We propose a novel algorithm for packet dropping in which the oldest packet is dropped (OPD). It will be shown that OPD provides fair service while having the minimal complexity.**

***Index Terms*—Internet router; packet dropping; scheduling; QoS;**

I. Introduction

Internet traffic keeps growing because of the increasing number of users and their bandwidth demands. As a consequence, link bit-rates and router capacities must increase to accommodate the growing traffic. At the same time, new applications are being developed, with diverse QoS requirements. Multimedia applications are getting more and more popular, and they require more stringent delay requirements than file transfers that dominated the Internet until recently.

However, bandwidth reservations are not yet implemented across the Internet on a large scale. Bandwidth reservations typically require some kind of centralization, which is difficult to implement in a scalable manner. Another issue is of administrative nature. Namely, there is a large number of network operators and internet service providers (ISPs) which have to agree on the common technology and protocols, in order to provide end-to-end QoS guarantees. Such an agreement is difficult to achieve. Internet grew fast thanks to its distributed nature, and now, it gets hard to coordinate different actors on the Internet more tightly. Therefore, best-effort (BE) traffic still dominates the Internet.

On the Internet, users still send their traffic without prior examination of the available resources, and negotiations with the network agents. Even multimedia contents are transferred as the BE traffic most of the time. If some links get congested, users adjust their rates according to the TCP protocol. But, TCP itself cannot provide fair allocation of the bandwidth among the traffic flows that pass through those links, since users might tweak their TCP codes. In addition, some applications do not use TCP. Consequently, it was recognized that routers and switches need to implement appropriate scheduling algorithms that isolate flows from each other and provide them the link portions which are roughly proportional to their weights [1]. Weights are administratively assigned based on the service level agreements (SLAs). One flow can be assigned to a particular user, company, ISP and so on.

The large number of fair scheduling algorithms have been proposed in the literature [2], [3]. These algorithms achieve different trade-offs between complexity and fairness level. Typically, the scheduling algorithms that achieve the higher level of fairness require more complex implementation. Scheduling algorithms are usually designed to provide max-min fairness, in which modest flows are entirely served if they comprise traffic that is smaller than their fair shares, while other flows share the bandwidth unused by the modest flows according to their weights. Deficit round-robin (DRR) is among the most popular max-min fair scheduling algorithms due to its low complexity, i.e. due to its practical value [4].

Fair scheduling algorithms specify the order in which the packets are sent through the outgoing link, but they do not specify how the packets are dropped. These algorithms might not be sufficient to ensure fair treatment if aggressive flows fill the buffer and packets of other flows get dropped as a consequence. Therefore, packets need to be dropped so that all the flows get fair access to the buffer before the link.

Buffers can be allocated to flows statically or dynamically. In the case of the static buffer allocation, which is often used in practice, a fixed slice of a buffer is allocated to each flow. When a fair algorithm is applied and the traffic of a flow exceeds its fair share, the buffer slice allocated to this flow will get full and the packets of the flow will be dropped. Static buffer allocation ensures fair shares of the link capacity to the flows, but might lead to an inefficient utilization of the buffer space. Namely, it may happen that some flows do not use their slices, which could have been used by aggressive flows. When the buffer space is allocated to the flows dynamically, flows can use arbitrary portions of the buffer space depending on their traffic load. In that case, the buffer is used efficiently, however, it may happen that aggressive flows monopolize the buffer space.

Packets can be dropped according to different algorithms. The simplest possible algorithm is tail drop (TD) in which packets coming to a full buffer are dropped. This algorithm is obviously simple to implement, however, it has certain disadvantages which will be explained in the paper. Random early detection (RED) algorithm is slightly more complicated. RED introduces two thresholds. Packets are dropped with the probability, that depends on the queue length, when their length is above the lower threshold, and with the probability equal to 1 when their length is above the higher threshold [5]. RED algorithm was shown to incur the smaller degree of synchronization among TCP flows and, therefore, the higher

efficiency. Also, it was shown to handle a bursty traffic more fairly. But later, advantages of RED were disputed through different set of scenarios [6].

Fairness also must be supported by the algorithms for packet dropping [1], [7]–[10]. One way to provide a fair access to the buffer is by dropping packets from the longest queues (LQD) as proposed initially in [1]. A variation of LQD was proposed in [9] where packets are dropped from longest queues with certain probabilities, only when the total queue length exceeds a certain threshold, in order to decrease the queuing delay. In a couple of solutions, the flow rates are measured, and when they exceed fair shares, the packets are dropped with certain probabilities [7], [10]. Finally, in [8], packets are dropped from the queues that exceed their allocated fair shares of the buffer. Most of the algorithms are complex because they include frequent calculation of fair shares that depends on the incoming traffic. Also, mentioned algorithms, except of TD and LQD, do not utilize the buffer fully.

In this paper, we propose a novel algorithm for packet dropping. It is very simple and practical. In our proposed algorithm the oldest packet is dropped when the buffer gets full. The oldest packet had most chances to be transmitted in the past according to the specified fair scheduling algorithm, and as such, it should be dropped. This algorithm will be named oldest packet drop (OPD). We will compare OPD with TD and LQD algorithms in terms of fairness, since all three of them utilize the buffers fully.

II. Algorithms for Packet Dropping

In this section, we will define precisely TD, LQD and OPD algorithms, and discuss their complexities. These three algorithms assume dynamic buffer allocation. This being the case, queues are implemented by linked lists in which each packet points to the packet of the same flow that arrived next after it. Two pointers are associated to each queue pointing to its starting and ending locations. Empty locations are organized in a separate linked list which has pointers to its starting and ending location too. Whenever, a packet arrives to or departs from some queue, its pointers are updated, as well as the pointers of the queue with empty locations.

Algorithm 1 Tail Drop

if $length(packet\_to\_rx) > buffer\_free\_space$ **then**
  $drop(packet\_to\_rx)$;
end if

TD algorithm is the simplest possible. When a new packet arrives, its length is learned from the header. If there is a sufficient space in the buffer, the packet is stored at the end of the appropriate queue. Otherwise, the incoming packet is dropped. Only one variable is needed to store the size of the empty space in the buffer. It is updated whenever a packet arrives, or departs the buffer.

LQD is also simple to describe, but it is more complex for implementation than TD. Similarly, as in TD, the length of an incoming packet is learned from its header. Then, it is determined if this packet can be stored in the buffer. If there is not enough space for the packet, then the oldest packets from the longest queue is dropped. It is again checked if there is enough space for the incoming packet, and if not, the described procedure is repeated until there is sufficient space to store the incoming packet. Incoming packet is stored at the end of the appropriate queue, when enough space becomes available. An array is needed to store lengths of all queues. It should be updated and sorted whenever a packet arrives or departs the buffer. Therefore, the complexity of the algorithm is roughly $O(log_2 F)$, where F is the number of flows.

Algorithm 2 Longest Queue Drop

while $length(packet\_to\_rx) > buffer\_free\_space$ **do**
  $longest\_queue := get\_longest\_queue()$;
  $packet\_to\_drop := oldest\_packet(longest\_queue)$;
  $buffer\_free\_space :=$
    $buffer\_free\_space + length(packet\_to\_drop)$;
  $drop(packet\_to\_drop)$;
end while
$receive(packet\_to\_rx)$;

Algorithm 3 Oldest Packet Drop

while $length(packet\_to\_rx) > buffer\_free\_space$ **do**
  $packet\_to\_drop := oldest\_packet$;
  $buffer\_free\_space :=$
    $buffer\_free\_space + length(packet\_to\_drop)$;
  $drop(packet\_to\_drop)$;
end while
$receive(packet\_to\_rx)$;

Finally, in our proposed OPD algorithm, if there is not sufficient space for an incoming packet in the buffer, the oldest packet is dropped. If the incoming packet does not fit the buffer even after the oldest packet is dropped, the next oldest packet is dropped. This process repeats until the incoming packet can be stored in the buffer. In order to be able to find quickly the oldest packet, packets need to be linked according to their times of arrival as well. So, each packet needs to point not only to the next packet of the same flow (queue), but also to the packet that came right before it, and to the packet that arrived just after it. Two additional pointers point to the oldest and the newest packets in the buffer. Whenever a packet departs the buffer, its preceding and succeeding packets need to update their pointers so that they point to each other. When a packet arrives to the buffer, only the previous packet needs to update its pointer, and when the oldest packet departs the buffer or is dropped, only its succeeding packet needs to update its pointer. All these pointer manipulations are simple, and the associated processing does not depend on the number of flows, i.e. the complexity of OPD is $O(1)$. Also, the memory space needed for two additional pointers per packet is negligible with respect to the packet size.

Pseudo codes for TD, LQD and OPD algorithms are shown

in Algorithms 1-3. As we explained, LQD algorithm is significantly more complex than TD and OPD algorithms. Complexities of TD and OPD algorithms do not depend on the number of flows.

III. Performance Comparison

TD, LQD and OPD algorithms were compared through time driven simulations of diverse scenarios. In all cases DRR scheduling algorithm is used [4]. The results of simulations are shown in Fig. 1-9. Parameters varied in these scenarios are the number of flows, F, probability of packet generation, P_1, probability that the generated packet will continue in the next slot, P_2, and flow weights, w.

In scenarios 1-3, 5 and 7, there are $G = 8$ groups with $F_G = 16$ consecutive flows, and, so, there are $F = 128$ flows. Other scenarios, 4, 6 and 8, assume $F = 1024$ flows, divided into $G = 64$ groups with $F_G = 16$ consecutive flows. Probabilities and weights of flows may vary in a group. Each flow is uniquely identified with variable f_{ID}, where $f_{ID} \in [1, F]$. Flows in different groups, whose identifications are the same modulo F_G, have the same probabilities and weights. The 3D graphs show for the selected set of flows the number of generated cells (input traffic), the number of served cells in the ideal case and in the cases when TD, LQD and OPD algorithms are applied. The buffer size is $B = 16K$ cells for all 3D graphics. Here, cell represents the amount of data transmitted within one time slot of the simulation. The 2D graphs show normalized flow throughputs as buffer size varies, for all three algorithms. Results are averaged for flows with the same parameters. Horizontal bars show normalized flow throughputs in the ideal case.

Scenario 1, shown in Fig. 1, comprises aggressive flows with the similar generated traffic loads, and different weights. In scenario 2, shown in Fig. 2, flows have the same weights and different traffic loads. Flows in these two scenarios have packets of different lengths, i.e. different P_2. In scenarios 3 and 4, shown in Fig. 3 and 4, all flows have the same weight and packets of equal length, but they have different traffic loads. These two scenarios differ only in number of flows. Scenario 3 has been tested for $F = 128$ flows, while scenario 4 assumes $F = 1024$ flows. Scenarios 5 and 6 show the performance when more aggressive flows have smaller weights, which can be observed in Fig. 5 and 6. Again, these two scenarios assume different numbers of flows, $F = 128$ flows, and $F = 1024$ flows, respectively. Finally, in scenarios 7 and 8, flows $f_{ID} \bmod F_G \in [5, 8]$ are modest and generate traffic loads smaller than their fair shares. We can see in Fig. 7 and 8 that these modest flows get all their traffic through, which is two times smaller than the capacity obtained by flows $f_{ID} \bmod F_G \in [1, 4]$ with the same weights.

In all scenarios, link shares allocated to the flows for the LQD and OPD algorithms are converging to the ideal case with the infinite buffer. In the case of the larger number of flows, $F = 1024$, OPD converges somewhat slower to the ideal case than LQD. So, OPD needs larger buffer to ensure fair bandwidth allocation. For $F = 1024$, the buffer storing $16K$ cells, is sufficient for perfect performance of OPD in all observed scenarios. Cells are typically 64 bytes in switches and routers, so, OPD has quite small memory requirements of 1MB to provide excellent fairness. On the other side, the TD algorithm does not provide fair shares to flows in any of the cases. This is because TD allows aggressive flows to monopolize the buffer space, not allowing other flows to access it. In scenarios 1 and 2, it was observed that flows with shorter packets gain larger than their fare shares when TD is used, while packet size does not matter in the case of LQD and OPD.

Finally, we observed the delay incurred by all three algorithms in scenarios 5 and 7 which are depicted in Fig. 9. TD provides various delays to different flows, where flows that are under-served experience smaller delays because their queues are shorter. On the other side, LQD and OPD algorithms provide similar delays to all the aggressive flows, and very small delay to the modest flows in scenario 7.

IV. Concluding Remarks

In this paper, we examined fairness provided by three algorithms for packet dropping. It turned out that our proposed algorithm OPD has the minimal complexity, while providing fairness. OPD converges to the ideal bandwidth allocation slower than LQD when the number of flows is large, while the simplest TD does not provide fairness. But, LQD is not scalable since its complexity increases with the number of flows.

Acknowledgement

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

References

[1] A. Demers, S. Keshav, and S. Shenker, "*Analysis and Simulation of a Fair Queueing Algorithm,*" ACM SIGCOMM Computer Communication Review, Vol. 19, no. 4, 1989.

[2] D. Nace, and M. Pioro, "*Max-min Fairness and its Applications to Routing and Load-Balancing in Communication Networks: A Tutorial,*" IEEE Communication Surveys and Tutorials, Vol. 10, no. 4, 2008.

[3] T. Bonald, L. Massoulie, A. Proutiere, and J. Virtamo, "*A Queueing Analysis of Max-Min Fairness, Proportional Fairness, and Balanced Fairness,*" Queueing Systems: Theory and Applications, Vol. 53, no. 1-2, 2006.

[4] M. Shreedhar, and G. Varghese, "*Efficient Fair Queueing Using Deficit Round Robin,*" ACM SIGCOMM Computer Communication Review, Vol. 25, no. 4, 1995.

[5] S. Floyd and V. Jacobson, "*Random Early Detection Gateways for Congestion Avoidance,*" IEEE/ACM Transactions on Networking, Vol. 1, no. 4, 1993.

[6] T. Bonald, M. May, and J. C. Bolot, "*Analytic Evaluation of RED Performance,*" Proc. IEEE INFOCOM 2000, Vol. 3, 2000.

[7] G. Aldabbagh, M. Rio, and I. Darwazeh, "*Fair Early Drop: An Active Queue Management Scheme for the Control of Unresponsive Flows,*" Proc. IEEE 10th ICCIT, 2010.

[8] T. Miyamura, K. Nakagawa, P. Dhananjaya, M. Aoki, and N. Yamanaka, "*Active Queue Control Scheme for Achieving Approximately Fair Bandwidth Allocation,*" Proc. IEEE ICC, Vol. 2, 2002.

[9] M. Nabeshima, and K. Yata, "*Performance Improvement of Active Queue Management with Per-flow Scheduling,*" IEEE Proc. on Communications, Vol. 152, no. 6, 2005.

[10] A. Racz, G. Fodor, and Z. Turanyi, "*Weighted Fair Early Packet Discard at an ATM Switch Output Port,*" Proc. IEEE INFOCOM 1999, Vol. 3, 1999.

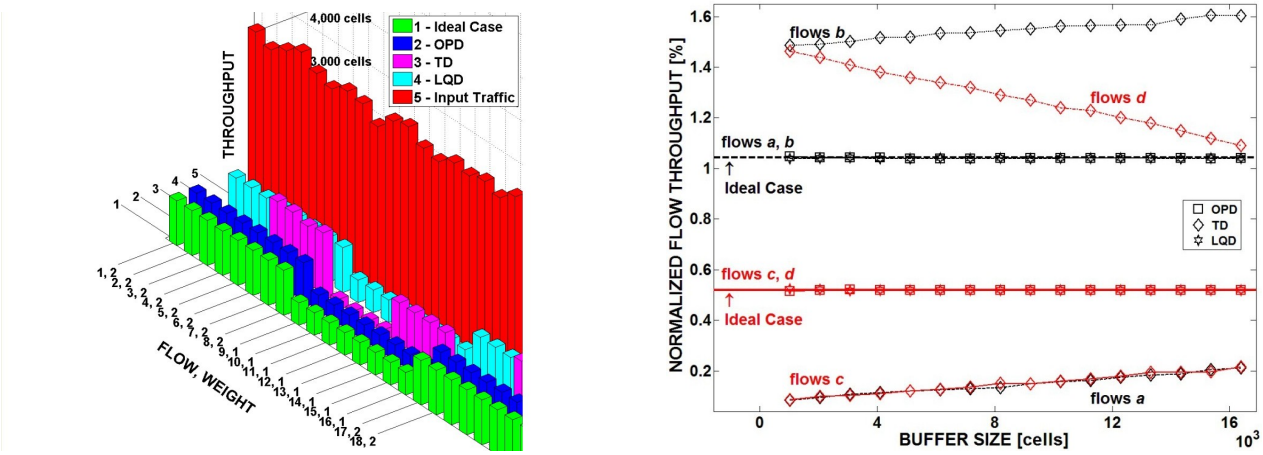


Fig. 1. Scenario 1: Black lines represent flows a and b, and red lines represent flows c and d.
$a \bmod F_G \in [1,4]$, $b \bmod F_G \in [5,8]$, $c \bmod F_G \in [9,12]$, $d \bmod F_G \in [13,16]$; F_G=16; F=128;
$P_1(a)$=$P_1(c)$=0.009, $P_1(b)$=$P_1(d)$=0.038; $P_2(a)$=$P_2(c)$=0.8, $P_2(b)$=$P_2(d)$=0.1; $w(a)$=$w(b)$=2, $w(c)$=$w(d)$=1.

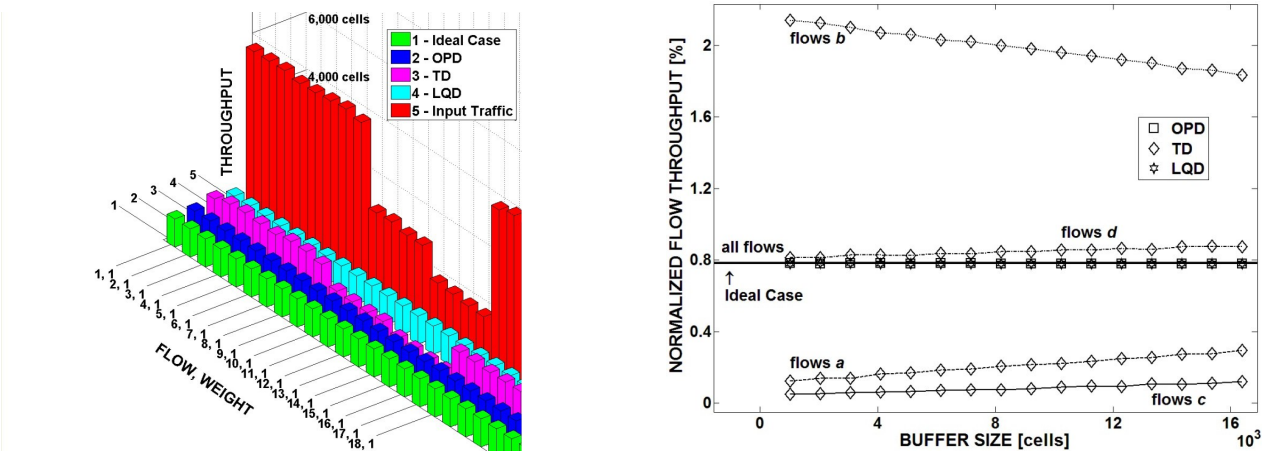


Fig. 2. Scenario 2: Dashed lines represent flows a, dotted lines represent flows b, solid lines represent flows c, and dash-dot lines represent flows d.
$a \bmod F_G \in [1,4]$, $b \bmod F_G \in [5,8]$, $c \bmod F_G \in [9,12]$, $d \bmod F_G \in [13,16]$; F_G=16; F=128; $P_1(a)$=0.013, $P_1(b)$=0.056, $P_1(c)$=0.049,
$P_1(d)$=0.021; $P_2(a)$=$P_2(c)$=0.8, $P_2(b)$=$P_2(d)$=0.1; $w(a)$=$w(b)$=$w(c)$=$w(d)$=1.

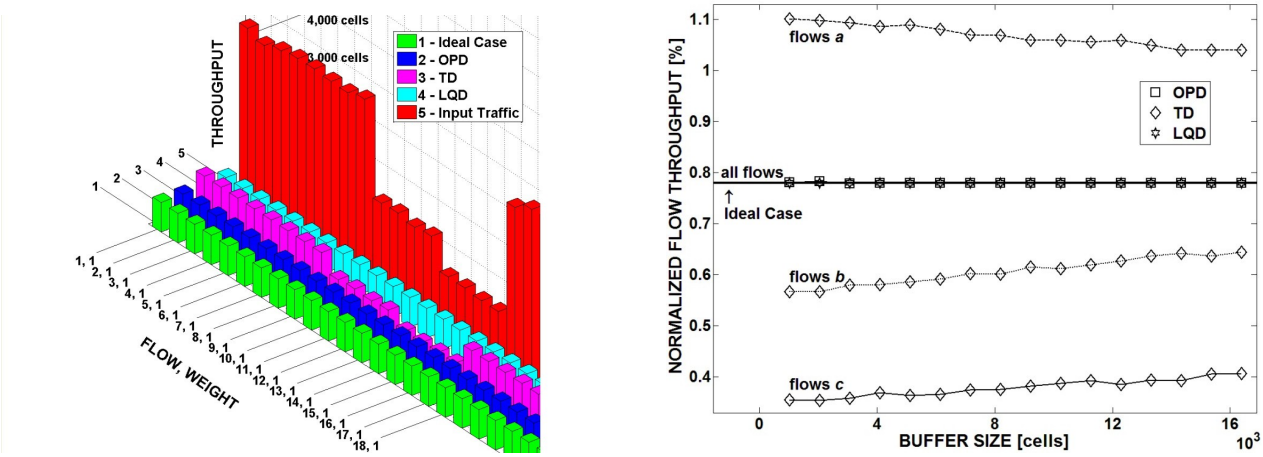


Fig. 3. Scenario 3: Dashed lines represent flows a, dotted lines represent flows b, and solid lines represent flows c.
$a \bmod F_G \in [1,8]$, $b \bmod F_G \in [9,12]$, $c \bmod F_G \in [13,16]$; F_G=16; F=128; $P_1(a)$=0.032, $P_1(b)$=0.016,
$P_1(c)$=0.01; $P_2(a)$=$P_2(b)$=$P_2(c)$=0.3; $w(a)$=$w(b)$=$w(c)$=1.

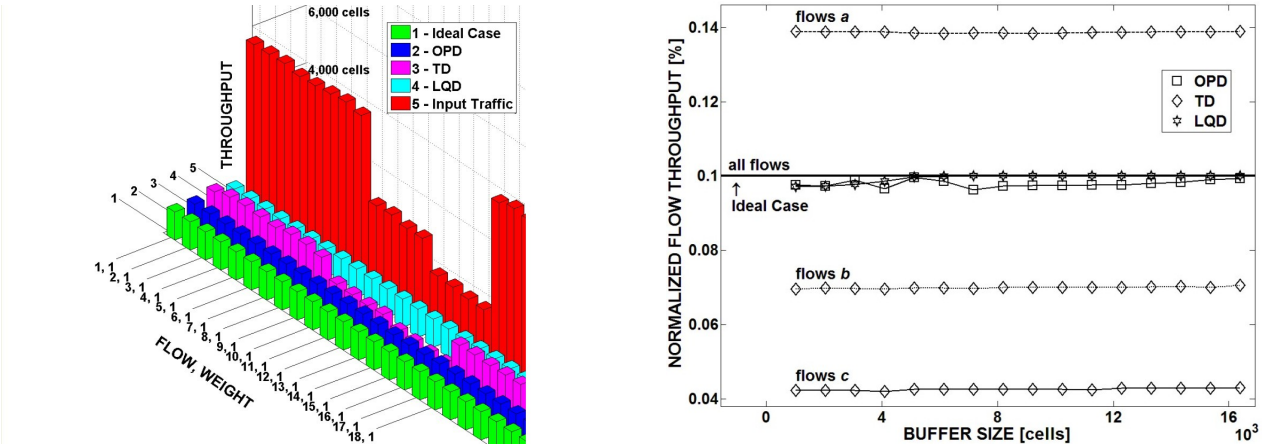


Fig. 4. Scenario 4: Dashed lines represent flows a, dotted lines represent flows b, and solid lines represent flows c. $a \bmod F_G \in [1,8]$, $b \bmod F_G \in [9,12]$, $c \bmod F_G \in [13,16]$; F_G=16; F=1024; $P_1(a)$=0.004, $P_1(b)$=0.002, $P_1(c)$=0.00125; $P_2(a)$=$P_2(b)$=$P_2(c)$=0.3; $w(a)$=$w(b)$=$w(c)$=1.

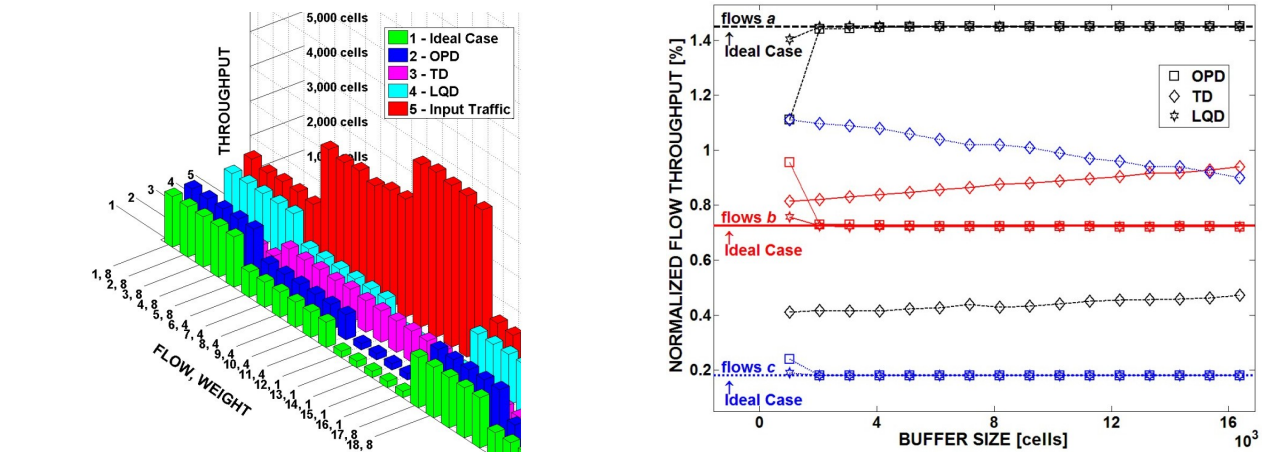


Fig. 5. Scenario 5: Dashed black lines represent flows a, solid red lines represent flows b, and dotted blue lines represent flows c. $a \bmod F_G \in [1,5]$, $b \bmod F_G \in [6,11]$, $c \bmod F_G \in [12,16]$; F_G=16; F=128; $P_1(a)$=0.01, $P_1(b)$=0.02, $P_1(c)$=0.028; $P_2(a)$=$P_2(b)$=$P_2(c)$=0.4; $w(a)$=8, $w(b)$=4, $w(c)$=1.

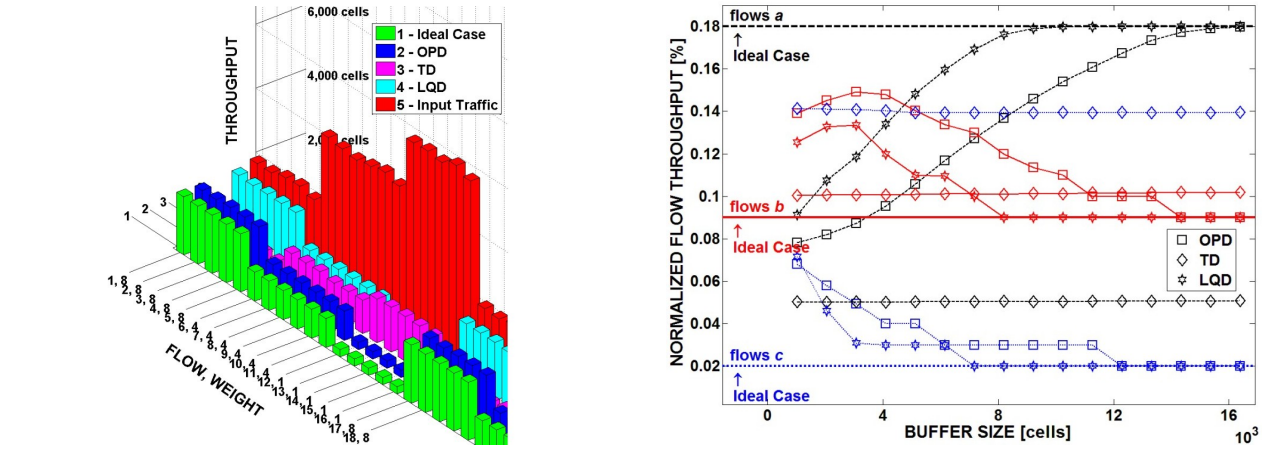


Fig. 6. Scenario 6: Dashed black lines represent flows a, solid red lines represent flows b, and dotted blue lines represent flows c. $a \bmod F_G \in [1,5]$, $b \bmod F_G \in [6,11]$, $c \bmod F_G \in [12,16]$; F_G=16; F=1024; $P_1(a)$=0.00125, $P_1(b)$=0.0025, $P_1(c)$=0.0035; $P_2(a)$=$P_2(b)$=$P_2(c)$=0.4; $w(a)$=8, $w(b)$=4, $w(c)$=1.

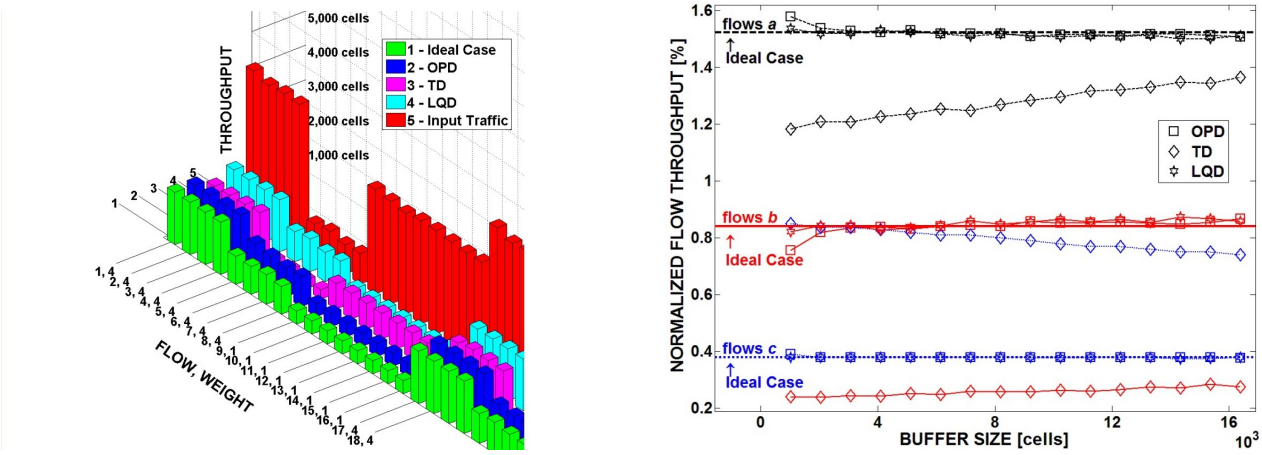


Fig. 7. Scenario 7: Dashed black lines represent flows a, solid red lines represent flows b, and dotted blue lines represent flows c. $a \bmod F_G \in [1,4]$, $b \bmod F_G \in [5,8]$, $c \bmod F_G \in [9,16]$; F_G=16; F=128; $P_1(a)$=0.025, $P_1(b)$=0.005, $P_1(c)$=0.018; $P_2(a)$=$P_2(b)$=$P_2(c)$=0.4; $w(a)$=8, $w(b)$=4, $w(c)$=1.

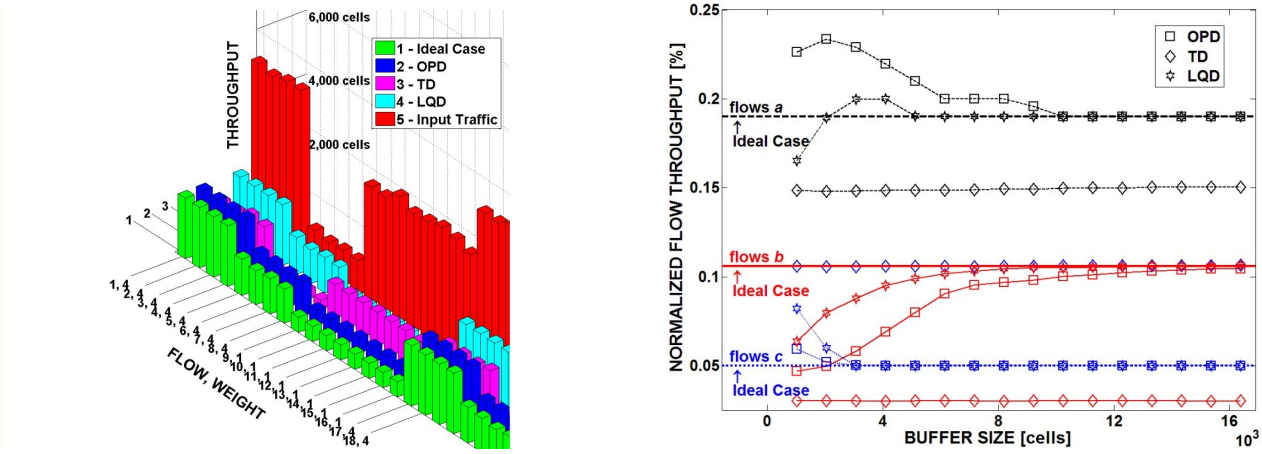
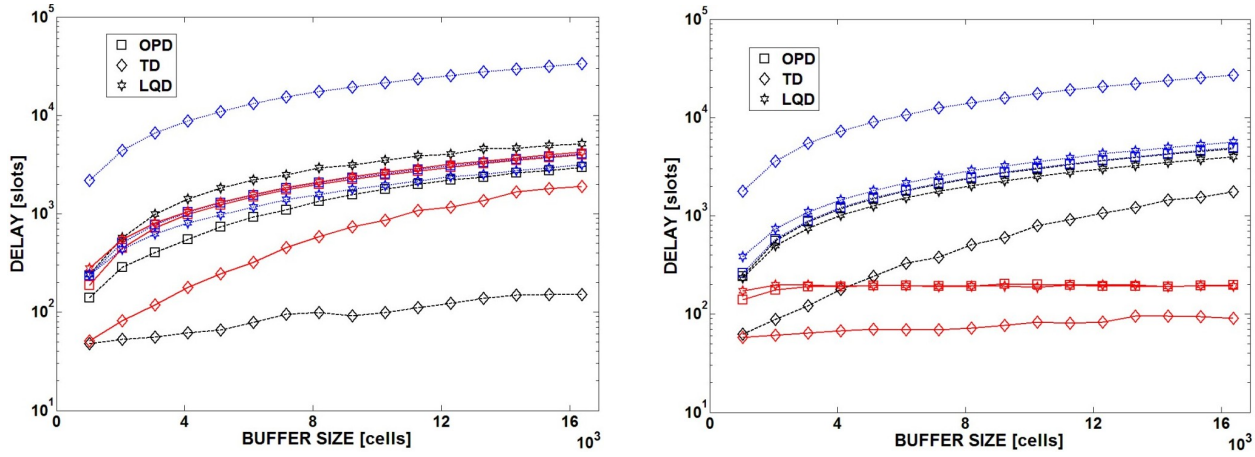


Fig. 8. Scenario 8: Dashed black lines represent flows a, solid red lines represent flows b, and dotted blue lines represent flows c. $a \bmod F_G \in [1,4]$, $b \bmod F_G \in [5,8]$, $c \bmod F_G \in [9,16]$; F_G=16; F=1024; $P_1(a)$=0.003125, $P_1(b)$=0.000625, $P_1(c)$=0.00225; $P_2(a)$=$P_2(b)$=$P_2(c)$=0.4; $w(a)$=8, $w(b)$=4, $w(c)$=1.



(a) Delays for scenario 5

(b) Delays for scenario 7

Fig. 9. Dashed black lines represent flows a, solid red lines represent flows b, and dotted blue lines represent flows c. For Fig. 9a flows a, b and c are defined in Fig. 5. For Fig. 9b flows a, b and c are defined in Fig. 7.