Technical Report : Study and Implementation of 20 Scheduling and Shaping Policies

Mostafa Elbediwy, Bill Pontikakis, Jean-Pierre David, and Yvon Savaria, *Fellow, IEEE*

***Abstract*—In this technical report, the pseudocodes for different scheduling policies, listed by the P4 working group, are presented. We are targeting to implement these algorithms in the ingress (enqueue) and egress (dequeue) pipelines of the P4 target switches. Between the Enqueue and the Dequeue levels, there are the DR-PIFO scheduler [1] [2], which always schedules the lowest rank eligible packet. The DR-PIFO scheduler is successfully integrated into target switches that are programmed by the P4 programming language. So, the following algorithms calculating the rank of packets/flows and/or the eligible time of packets. In case of shaping policies, a packet should not be scheduled before its eligible time.**

In this work, we present 20 scheduling and shaping policies. 15 of them already have a well-defined rank-based version which is compatible with the DR-PIFO scheduler. the other 5 policies which are : Deficit Round Robin (DRR), Weighted Round Robin (WRR), Weighted Deficit Round Robin (WDRR), Slytherin, and the rate-limited Stricty Priority (non-work-conerving), require novel rank-based versions to utilize the DR-PIFO scheduler. Slytherin is the only scheduling policies out of these 5 policies that have an approximated rank-based version which seems to provide an unfair service to the network users. Accordingly, in a companion paper, we propose novel rank-based versions for these 5 scheduling policies while, in this report, we only present the original algorithms of these policies in Algorithms 1 2 3 4 5.

Algorithm 1: (P1) Original Deficit Round Robin [3]

```
/* In DRR, there are multiple "FIFO" queues,
   one dedicated to each flow.
   "Quantum" is the maximum number of bytes,
   that a flow can send every round.
   N : is the number of the supported flows.  */
At Enqueue of each new packet (P) :
  /* Insert the new packet at the tail of the
     FIFO queue of its flow.                  */
  FIFO[P.flow_id].enqueue(P);

Dequeueing Module :
while true do
  for i ← 1 to N do
    if FIFO[i] not empty then
      deficit_count[i] += Quantum
      while deficit_count[i] >= Size(FIFO[i].head) and FIFO[i] not empty do
        deficit_count[k] -= Size(FIFO[i].head);
        FIFO[i].dequeue();
      if FIFO[i] is empty then
        deficit_count[i] = 0
```

Algorithm 2: (P2) Original Weighted Round Robin [4]

```
/* "Weight" is the maximum number of packets,
   that a flow can send every round.          */
At Enqueue of each new packet (P) :
  FIFO[P.flow_id].enqueue(P);

Dequeueing Module :
while true do
  for i ← 1 to N do
    sent_pkts = 0
    while sent_pkts < weight[i]) and FIFO[i] not empty do
      ++ sent_pkts;
      FIFO[i].dequeue();
```

Algorithm 3: (P3) Original Weighted Deficit Round Robin [5][6]

```
At Enqueue of each new packet (P) :
    FIFO[P.flow_id].enqueue(P);
Dequeueing Module :
while true do
    for i ← 1 to N do
        if FIFO[i] not empty then
            deficit_count[i] += Quantum[i]
            while deficit_count[i] >=
              Size(FIFO[i].head) and FIFO[i] not empty
              do
                deficit_count[k] -=
                  Size(FIFO[i].head);
                FIFO[i].dequeue();

            if FIFO[i] is empty then
                deficit_count[i] = 0
```

Algorithm 4: (P4) Original Slytherin [7]

```
At Enqueue of each new packet (P) :
if CE bit is set then then
    P.rank = low_rank;  // high priority
else
    P.rank = high_rank;  // low priority
```

Algorithm 5: (P5) Original Rate Limited Strict Priority (Non-Work-Conserving)

```
Dequeueing Module :
while true do
    for Q ← high_queues to low_queues do
        if Q not empty and
          current_rate(Q) < rate_limit(Q) then
            Q.dequeue();
```

Algorithm 6: (P6) Least Attained Service (LAS) [8]

```
// array to count the service received by each flow.
        service_count = array[number_of_flows];

  At Enqueue of each new packet (P) :
        service_count[P.flow_id] =
                service_count[P.flow_id] + P.length;
        P.rank = service_count[P.flow_id];
```

Algorithm 7: (P7) pFabric with starvation prevention

```
current_rank = array[N];

  At Enqueue of each new packet (P) :
P.rank = P.remaining_flow_size;
if P.rank < current_rank[P.flow_id] or
  current_rank[P.flow_id] == 0 then
    current_rank[P.flow_id] = P.rank;
    update_rank = current_rank[P.flow_id];
    update_id = P.flow_id;
```

Algorithm 8: (P8) Weighted Fair Queueing (WFQ) [9] [10]

```
// array for the weights assigned to each flow.
        weights = array[number_of_flows];

  // array for the finish time for each flow
        finish_time = array[number_of_flows];

  At Enqueue of each new packet (P) :
    length_over_weight = P.length / weights[P.flow_id];

  if finish_time[P.flow_id] > P.arrival_time then
    finish_time[P.flow_id] = finish_time[P.flow_id]
              + length_over_weight;
else
    finish_time[P.flow_id] = P.arrival_time
              + length_over_weight;

    P.rank = finish_time[P.flow_id];
```

Algorithm 9: (P9) Least Slack Time First (LSTF) [11]

```
At Enqueue of each new packet (P) :
        P.rank = P.slack + P.arrival_time;

  At Dequeue of each scheduled packet (P) :
        P.slack = P.rank - P.departure_time;
// if a packet queued for more than its slack, it can be
// dropped.
```

Algorithm 10: (P10) Least Attained Recent Service (LARS) [12]

// array to count the service received by each flow.
attained_service = **array**[number_of_flows];

At Enqueue of each new packet (P) :
if *current_time_unit* $>=$ *decay_time_unit* **then**
attained_service[P.flow_ID] =
attained_service[P.flow_ID] * decay_factor;
++ current_time_unit;
attained_service[P.flow_ID] =
attained_service[P.flow_ID] + P.length;
P.rank = attained_service[P.flow_ID];

// To avoid starvation, the ranks of previous packets
// may need to be updated if their ranks are higher
// than the new packet's rank.

Algorithm 11: (P11) Stop-and-Go [13]

// T : the length of the time frames
At Enqueue of each new packet (P) :
if $current\_time >= frame\_end\_time$ **then**
frame_begin_time = frame_end_time;
frame_end_time = frame_begin_time + T;
P.rank = frame_end_time;

Algorithm 12: (P12) Rate Controlled Service Disciplines (RCSD), (Jitter-EDD)

At Enqueue of each new packet (P) :
P.rank = P.local_deadline + P.arrival_time;
P.eligible_time = P.Ahead + current_time;

At Dequeue of each scheduled packet (P) :
if *P.rank* $<$ *P.departure_time* **then**
P.Ahead = P.departure_time - P.rank;
else
P.Ahead = 0;

Algorithm 13: (P13) Window Constrained Scheduling - Virtual Deadline Scheduling (VDS) [14] [15]

// Constraints parameters.
M = **array**[number_of_flows];
K = **array**[number_of_flows];
T = **array**[number_of_flows];
$M^{'}$ = **array**[number_of_flows];
$K^{'}$ = **array**[number_of_flows];
// the arrival time of the last packet from each flow.
arrival_times = **array**[number_of_flows];

At Enqueue of each new packet (P) :
arrival_times[P.flow_id] = P.arrival_time;
$--K^{'}$[P.flow_id];
if $K^{'}$ *[P.flow_id]* $==$ *0* **then**
// if $M^{'}$ is above zero, that could be a violation.
$K^{'}$[P.flow_id] = K[P.flow_id];
$M^{'}$[P.flow_id] = M[P.flow_id];

P.rank = ((T[P.flow_id] * $K^{'}$[P.flow_id]) /
$M^{'}$[P.flow_id]) + P.arrival_time;
for *all previous packets (PP) with the same P.flow_id* **do**
PP.rank = P.rank;

At Dequeue of each scheduled packet (P) :
// assume synchronization reset of $M^{'}$ and $K^{'}$
$--M^{'}$[P.flow_id];
if $M^{'}$ *[P.flow_id]* $==$ *0* **then**
$M^{'}$[P.flow_id] = M[P.flow_id];
new_rank = ((T[P.flow_id] * $K^{'}$[P.flow_id]) /
$M^{'}$[P.flow_id]) + arrival_times[P.flow_id];
for *all previous packets (PP) with the same P.flow_id* **do**
PP.rank = new_rank;
// Note : in case of violation, packets may be dropped.

Algorithm 14: (P14) Approximate Fair Queueing [16]

"srv_cntr" : array to count the service received by each flow.
"Q" : "Quantum" the maximum number of bytes, that a flow can send every round.
At Enqueue of each new packet (P) :
srv_cntr[P.flow_id] **+=** P.length;
round_id = $\lfloor(srv\_cntr[P.flow\_id]-1)/Q\rfloor$;
P.rank = round_id;

Algorithm 15: (P15) Hierarchical Policies (FIFO-DRR-SP)

```
// As an example :
// Assume 3-level hierarchy,
// level-2 (root) Strict-Priority (SP) scheduling,
// level-1 (middle) Weighted Fair Queueing (WFQ),
// level-0 (leaf) First-In-First-Out (FIFO) scheduling.

// array for the weights assigned to each flow.
        weights = array[number_of_flows];

  // array for the finish time for each flow
        finish_time = array[number_of_flows];

  At Enqueue of each new packet (P) :
// FIFO ordering
        P.rank[0] = P.arrival_time;
// WFQ
    length_over_weight = P.length / weights[P.flow_id];
if finish_time[P.flow_id] > P.arrival_time then
    finish_time[P.flow_id] = finish_time[P.flow_id]
            + length_over_weight;
else
    finish_time[P.flow_id] = P.arrival_time
            + length_over_weight;
    P.rank[1] = finish_time[P.flow_id]; // SP (ToS, is
  the priority class for this packet).
    P.rank[2] = P.ToS; // assign high or low priority
```

Algorithm 16: (P16) Penalize Heavy Hitters (PHH) [17]

```
// array to count the number of received packets
// from each flow in the current time window.
        received_packets = array[number_of_flows];

// the time of the last reset triggered for each flow id
        last_reset_time = array[number_of_flows];

At Enqueue of each new packet (P) :
if (current_time - last_reset_time[P.flow_id]) >=
  window_period then
        last_reset_time[P.flow_id] = current_time;
        received_packets[P.flow_id] = 0;


        ++ received_packets[P.flow_id];
if received_packets[P.flow_id] >= Threshold then
    // Low priority
        P.rank = Higher_Rank_Value;
else
    // High priority
        P.rank = Lower_Rank_Value;
```

Algorithm 17: (P17) WFQ with weights determined by queue occupancy (WFQ-QO) [18]

```
// array for the expected ratio between the delay of
// each flow versus the delays of the rest of the flows.
        delay_ratio = array[number_of_flows];

// array to store the weights of each flow.
        weight = array[number_of_flows];

// array to calculate the number of stored packets
// from each flow.
        queue_occupancy = array[number_of_flows];

// array for the finish time for each flow
        finish_time = array[number_of_flows];

  // the time of the last reset triggered for each flow id
        last_reset_time = array[number_of_flows];

At Enqueue of each new packet (P) :
if (current_time - last_reset_time[P.flow_id]) >=
  window_period then
    weight[P.flow_id] = queue_occupancy[P.flow_id] /
                        delay_ratio[P.flow_id];
    last_reset_time[P.flow_id] = current_time;


    length_over_weight = P.length / weights[P.flow_id];

if finish_time[P.flow_id] > P.arrival_time then
    finish_time[P.flow_id] = finish_time[P.flow_id]
            + length_over_weight;
else
    finish_time[P.flow_id] = P.arrival_time
            + length_over_weight;


    ++ queue_occupancy[P.flow_id];
    P.rank = finish_time[P.flow_id];



At Dequeue of each scheduled packet (P) :
    −− queue_occupancy[P.flow_id];

// Note : to implement the ”Dynamic WFQ”
// scheduling algorithm, instead of only
// (queue_occupancy), it should be replaced by
// ( queue_occupancy + arrival_rate[P.flow_ID] ).
```

Algorithm 18: (P18) Rate Limited Strict Priority and Work-Conserving (RL-SP-WC)

```
// array to count the number of received packets
// from each flow in the current time window.
        received_packets = array[number_of_flows];

// the current window ID of each flow.
        window_ID = array[number_of_flows];

// array contains the maximum packets that each
// flow can send during a single time window.
        maximum_packets = array[number_of_flows];

// number of possible ranks at a single time window.
    possible_ranks = number_of_possible_SP_ranks *
            \sum_{n=0}^{n<number_of_flows} maximum_packets[n]

At Enqueue of each new packet (P) :
if (received_packets[P.flow_ID] ==
  maximum_packets[P.flow_ID]) then
        received_packets[P.flow_ID] = 0;
        ++ window_ID[P.flow_id];

        ++ received_packets[P.flow_id];
// P.ToS is the strict priority value for each packet.
        P.rank = P.ToS + window_ID[P.flow_id] *
                                    possible_ranks;
```

Algorithm 19: (P19) Variant of Weighted Fair Queueing (NUMFabric) [19]

```
// array for the finish time for each flow
        finish_time = array[number_of_flows];

  At Enqueue of each new packet (P) :
// weights are dynamically updated by the received
// packets. maybe a register is needed, if weights are
// usually updated once in a while.
    length_over_weight = P.length / P.weight;

  if finish_time[P.flow_id] > P.arrival_time then
    finish_time[P.flow_id] = finish_time[P.flow_id]
              + length_over_weight;
else
    finish_time[P.flow_id] = P.arrival_time
              + length_over_weight;

     P.rank = finish_time[P.flow_id];
```

Algorithm 20: (P20) Independent Scheduling and Shaping Policies (ISSP)

```
/* Example : 2-level hierarchy, level-1 (root,
   SP) and level-2 (leaf, WFQ). In addition,
   rate-limiting is applied at level-2. */
weights = array[N];
finish_time = array[N];
srv_cntr = array[N];
pkts_per_rnd = array[N];
window_ID = array[N];
 At Enqueue of each new packet (P) :
// rate-limiting
if srv_cntr[P.flow_ID] == pkts_per_rnd[P.flow_ID]
then
    ++ window_ID[P.flow_ID];
    srv_cntr[P.flow_ID] = 0;
++ srv_cntr[P.flow_ID];
P.eligible_time = window_ID[P.flow_ID] *
                                  window_period;
length_over_weight = P.length / weights[P.flow_id];
if finish_time[P.flow_id] > P.arrival_time then
    finish_time[P.flow_id] = finish_time[P.flow_id]
                            + length_over_weight;
else
    finish_time[P.flow_id] = P.arrival_time
                            + length_over_weight;
P.rank[0] = finish_time[P.flow_id]; // WFQ
P.rank[1] = P.ToS; // SP
```

References

[1] M. Elbediwy, B. Pontikakis, A. Ghaffari, J.-P. David, and Y. Savaria, “Dr-pifo: A dynamic ranking packet scheduler using a push-in-first-out queue,” *IEEE Transactions on Network and Service Management*, pp. 1–1, 2023.

[2] M. Elbediwy, B. Pontikakis, J.-P. David, and Y. Savaria, “A hardware architecture of a dynamic ranking packet scheduler for programmable network devices,” *IEEE Access*, 2023.

[3] M. Shreedhar and G. Varghese, “Efficient fair queuing using deficit round-robin,” *IEEE/ACM Transactions on networking*, vol. 4, no. 3, pp. 375–385, 1996.

[4] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, “Weighted round-robin cell multiplexing in a general-purpose atm switch chip,” *IEEE Journal on selected Areas in Communications*, vol. 9, no. 8, pp. 1265–1279, 1991.

[5] J. Lakkakorpi, A. Sayenko, and J. Moilanen, “Comparison of different scheduling algorithms for wimax base station: Deficit round-robin vs. proportional fair vs. weighted deficit round-robin,” in *2008 IEEE Wireless Communications and Networking Conference*. IEEE, 2008, pp. 1991–1996.

[6] R. Sharmal, S. S. Sehra, and S. K. Sehra, “Review of different queuing disciplines in voip, video conferencing and file transfer,” *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 4, no. 3, pp. 264–267, 2015.

[7] H. Rezaei, M. Malekpourshahraki, and B. Vamanan, “Slytherin: Dynamic, network-assisted prioritization of tail packets in datacenter networks,” in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2018, pp. 1–9.

[8] I. A. Rai, E. W. Biersack, and G. Urvoy-Keller, “Size-based scheduling to improve the performance of short tcp flows,” *IEEE network*, vol. 19, no. 1, pp. 12–17, 2005.

[9] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queueing algorithm,” *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 4, pp. 1–12, 1989.

[10] K. Parekh, Abhay and G. Robert, G., “A generalized processor sharing approach to flow control in integrated services networks: the single-node case,” *IEEE/ACM transactions on networking 1*, no. 3, pp. 344–357, 1993.

[11] J. Y.-T. Leung, “A new algorithm for scheduling periodic, real-time tasks,” *Algorithmica*, vol. 4, no. 1-4, p. 209, 1989.

[12] M. Heusse, G. Urvoy-Keller, A. Duda, and T. X. Brown, “Least attained recent service for packet scheduling over wireless lans,” in *2010 IEEE International Symposium on” A World of Wireless, Mobile and Multimedia Networks”(WoWMoM)*. IEEE, 2010, pp. 1–9.

[13] S. J. Golestani, “A stop-and-go queueing framework for congestion management,” in *Proceedings of the ACM symposium on Communications architectures & protocols*, 1990, pp. 8–18.

[14] Y. Zhang and R. West, “End-to-end window-constrained scheduling for real-time communication,” in *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA’04)*, 2004.

[15] Y. Zhang, R. West, and X. Qi, “A virtual deadline scheduler for window-constrained service guarantees,” in *25th IEEE International Real-Time Systems Symposium*. IEEE, 2004, pp. 151–160.

[16] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, “Approximating fair queueing on reconfigurable switches,” in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018, pp. 1–16.

[17] L. Yang, B. Ng, and W. K. Seah, “Heavy hitter detection and identification in software defined networking,” in *2016 25th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2016, pp. 1–10.

[18] C.-C. Li, S.-L. Tsao, M. C. Chen, Y. Sun, and Y.-M. Huang, “Proportional delay differentiation service based on weighted fair queuing,” in *Proceedings Ninth International Conference on Computer Communications and Networks (Cat. No. 00EX440)*. IEEE, 2000, pp. 418–423.

[19] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, “Numfabric: Fast and flexible bandwidth allocation in datacenters,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 188–201.