Spring: Theory and an Efficient Heuristic for Programmable Packet Scheduling with SP-PIFO

Balázs Vass, *Member, IEEE*

***Abstract*—Theoretical hardware model Push-In First-Out (PIFO) is used for programmable packet scheduling, allowing for flexible and dynamic reconfiguration of scheduling policies. SP-PIFO (Strict Priority-PIFO), on the other hand, is a practical emulation of PIFO that can be easily implemented using standard P4 switches. The efficiency of SP-PIFO relies on a heuristic called Push-Up/Push-Down (PUPD), which dynamically adjusts the mapping of input packets to a fixed set of priority queues in order to minimize scheduling errors compared to an ideal PIFO. This paper presents the first formal analysis of the PUPD algorithm. Our analysis shows that as more priority queues are added to the system, the ability of PUPD to emulate an optimal PIFO model decreases linearly. Based on this finding, we propose an optimal offline scheme that can determine the optimal SP-PIFO configuration in polynomial time, given a stochastic model of the input. Additionally, we introduce an online heuristic called Spring, which aims to approximate the offline optimum without requiring a stochastic input model. Our simulations demonstrate that Spring can improve the performance of SP-PIFO by a factor of 2x in certain configurations.**

***Index Terms*—programmable packet scheduling, P4, SP-PIFO, competitive analysis, polynomial algorithm, online heuristic**

1. Introduction

Traditionally, fixed-function switches implement specific network protocols engraved into the hardware. More recently, programmable switches have emerged, allowing network operators to refine on-the-fly the packet processing functionality, including header parsing and forwarding policies, using a high-level programming language like P4 [2]. However, packet scheduling in P4 switches has remained mostly fixed until recently. Push-In First-Out (PIFO) was the first hardware abstraction that, theoretically, enabled programming new scheduling algorithms into a switch without changing the hardware layout [3], [4]. In PIFO, each packet is assigned a *rank*, and the switch maintains packets in the hardware queues sorted by their rank (see Fig. 1). Then, different scheduling policies, like SRPT (Shortest Remaining Processing Time) or STFQ (Start-Time Fair Queueing) [4], can be implemented on top of PIFO by changing the way ranks are assigned to packets.

Lacking a viable hardware realization, PIFO had been considered mostly a theoretical possibility until the appearance of Strict Priority PIFO (SP-PIFO, [5]). SP-PIFO approximates PIFO by maintaining a set of strict priority (SP) queues and dynamically adapting the mapping between packet ranks and SP queues. The objective is to minimize the number *inversions*, where inversions connote the event that a high-rank (i.e., 'low-importance') packet precedes a low-rank (i.e., 'very important') packet during dequeuing. As such, the inversion count effectively measures the rate of 'scheduling mistakes' relative to an ideal PIFO implementation (see Fig.1 again). SP-PIFO maps packets to queues by maintaining a *queue bound* for each queue, which determines the smallest packet rank that can be assigned to the queue. Then, the mapping is adapted by dynamically adjusting the queue bounds in concert with the ingress traffic ranks using the Push-Up/Push-Down (PUPD) heuristic (see later). PUPD can be implemented in P4 and thus can run inside the data plane at line rate.

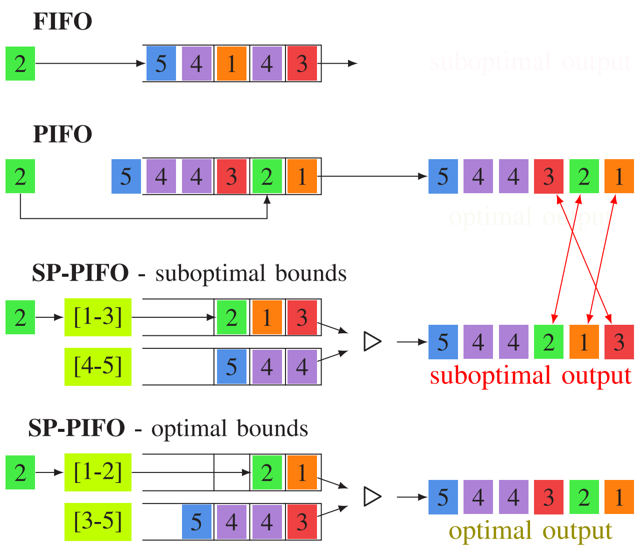


Fig. 1: Programmable scheduling: FIFO, PIFO, and SP-PIFO. The packets arrived in the order they are enqueued in FIFO (right to left). Trivially, FIFO does not sort, while PIFO releases the smallest ranked packets first. To avoid unnecessary rank inversions, the queue bounds of SP-PIFO must be optimised.

The efficiency of SP-PIFO is ultimately contingent on the capacity of PUPD to adapt the way packets are assigned to queues *quickly* and *efficiently*. Unfortunately, as of now, no thorough formal analysis of PUPD is available, nor is it known whether more efficient algorithms could be defined to drive SP-PIFO. Our primary goal in this study is to fill this gap.

The main contributions are as follows. After a quick recap on programmable scheduling (Sec. 2), we present the first *competitive analysis of PUPD* (Sec. 3). Our results are mostly

B. Vass is with High Speed Networks Laboratory (HSNLab), Department of Telecommunications and Media Informatics, Faculty of Electrical Engineering and Informatics (VIK), Budapest University of Technology and Economics, and Faculty of Mathematics and Computer Science, Babes-Bolyai University, Cluj Napoca, Romania. (E-mail: balazs.vass@tmit.bme.hu.) An earlier version of this paper appeared on IEEE INFOCOM Workshop on Networking Algorithms (WNA) 2022 [1].

DOI: 10.36244/ICJ.2025.3.1

negative: we show that the number of inversions produced by PUPD can be n times worse than that of a hypothetical 'clairvoyant' optimal bound-adaptation scheme, where n is the number of SP queues. This result suggests that the capacity of PUPD to adapt to yet unknown future ranks is limited, compared to an ideal bound-adaptation algorithm that 'knows the future', and the optimality gap increases linearly with the number of queues in SP-PIFO.

Driven by this observation, our next goal is to define an optimal algorithm. Our result in this context is a *polynomial-time offline algorithm*, which, given the probability distribution of ranks in incoming packets, computes a set of optimal queue bounds to minimize the number of inversions (Sec. 4).

Given that the rank distribution is hard to fix offline, our third contribution is a *fast online algorithm* that dynamically 'learns' the rank distribution and adaptively adjusts the SP-PIFO queue bounds following the learned distribution (Sec. 5).[1] Our evaluations (Sec. 6) suggest that the new algorithm can approximate the optimal queue bounds more efficiently than PUPD. We conclude the paper in Sec. 8.

2. Background

Suppose input packet ranks are taken from the interval $[1, k]$ and assume we are given n SP queues $Q[i] : i \in [1, n]$. The main idea in SP-PIFO is to maintain a queue bound q_i with respect to each queue so that all packets with rank from q_i to $q_{i+1} - 1$ are assigned to $Q[i]$ (assuming an imaginary q_{n+1} equaling $k + 1$), and adapt the bounds q_i as follows:

- At the beginning, all queue bounds are 0: $\forall i \in [1, n] : q_i = 0$.
- An incoming packet with rank r is enqueued into queue $Q[i]$ if $r \geq q_i$ and i is maximal among the queues with this property, and q_i is immediately set to r (*push-up*).
- If no such i exists (i.e., $p < q_1$), then r is enqueued to $Q[1]$ and all queue bounds are decreased by $q_1 - r$ (*push-down*).
- SP queues are drained in strict priority order: packets from $Q[1]$ are dequeued first; if $Q[1]$ is empty, then $Q[2]$ is dequeued next, etc.

PUPD is appealing for a number of reasons. First, it closely emulates the ideal PIFO behaviour in that it tends to assign low-rank (i.e., 'important') packets to high-priority queues and high-rank (i.e., 'low importance') packets to low-priority queues so that the packet sequence leaving the SP queues is approximately sorted by rank. Second, PUPD can be implemented in P4 entirely, maintaining the queue bounds in P4 registers. Hence, bound adaptation in PUPD occurs after processing each packet, at line rate. Third, PUPD can dynamically adapt the bounds to even potentially rapidly changing input rank patterns. Experimental evaluations show that PUPD can produce consistently good performance under a wide range of operational conditions [5].

The efficiency of SP-PIFO to emulate an optimal PIFO model is contingent on its capacity to map low-rank packets to high-priority queues consistently, and this is ultimately dependent on the efficiency of PUPD to adapt queue bounds to prevent 'scheduling mistakes'. Here, any deviation of the output sequence of SP-PIFO from an ideal PIFO sequence, which is strictly sorted by packet rank, is considered an error.

[1]Compared to the former version [1], the paper now includes a proof of (exponential) stability of a theoretical algorithm in a somewhat relaxed problem setting. The algorithm is very close to our offered Sring heuristic.

3. A competitive analysis of PUPD

Due to the limited lookahead available in the P4 data plane pipeline, SP-PIFO bound-adaptation is severely hampered by the unpredictability of the ranks of future packets. Competitive analysis is a methodology to analyze such *online* algorithms by comparing the performance to an optimal *offline* scheme that can access the entire sequence of future requests in advance. The *competitive ratio* is defined as the quotient of the worst-case error produced by the online and the offline algorithm over an adversarial input. The smaller the competitive ratio, the less the online algorithm is hampered by the unavailability of future requests and the closer the algorithm is to 'optimality'. Contrariwise, a relatively huge competitive ratio suggests that the performance penalty of the online setting can be prohibitive. The results below suggest that for PUPD, this is indeed the case in the case of *deterministic* and *stochastic* input packet sequences alike.

A. Deterministic competitive ratio

Given a *deterministic packet sequence* S, we measure the total error over S as the number of inversions via a simplified metric that enumerates only the intra-queue inversions, defined as follows:

$$\begin{aligned} U_{\texttt{det}}(S) := \#(\{a, b\} \subseteq S : a < b \text{ and} \\ a \text{ and } b \text{ enqueued to the same } Q[i] \text{ and} \\ a \text{ is the next packet enqueued in } Q[i] \text{ after } b) \ . \end{aligned} \tag{1}$$

We will show that there is a family of packet sequences S, for which PUPD produces n times more inversions than the optimal offline algorithm in terms of the error function $U_{\texttt{det}}(S)$, where n is the number of SP queues. Intuitively, PUPD is getting further from the optimum as we add more queues, not being able to fully utilize the additional flexibility yielded by a higher number of queues. Unfortunately, this is the case even if the number of distinct ranks k is just one more than n (for $k = n$, the problem can be solved trivially by assigning each rank to a separate queue, ordered priority-wise; PUPD is intuitively close to this solution, as no push-down can happen).

Theorem 1: Given n queues and $k \geq n + 1$ input ranks, the competitive ratio of PUPD is at least n in terms of error function (1).

Proof: In our construction, we will set the maximum rank to $k := n + 1$. We will use an arbitrary positive integer l. Let the input rank sequence be as follows (with the first packet to arrive on the right):

$$S = l \times [n, \ldots, 2, 1, 2, \ldots, n, n + 1] \ .$$

Here, given a packet sequence `[Seq]`, sequences $l \times$ `[Seq]` denotes the l-time repetition of `[Seq]`.

With this, the queues built by the PUPD are as follows (first packet enqueued to and dequeued from right):

$$Q_{\texttt{PUPD}}[i] = l \times [i, i+1], \qquad \forall i \in \{1, \ldots, n\}.$$

This means a number of l inversions in each queue, that sums up to nl for the PUPD.

On the other hand, by setting constant offline queue bounds to $\underline{q} = [q_1 = 2, q_2 = 3, \ldots, q_n = n+1]$, the packet sequences built in the queues are:

$$\begin{aligned} &Q_{\texttt{offline}}[1] = l \times [2,1,2], \\ &Q_{\texttt{offline}}[i] = (2l) \times [i+1], \qquad \forall i \in \{2, \ldots, n-1\}, \\ &Q_{\texttt{offline}}[n] = l \times [n+1]. \end{aligned}$$

In this offline setting, we have l inversions in the first queue, and none in the other queues.

The number of inversions made by the PUPD divided by those made by the offline algorithm is n for any value of l. This means that the competitive ratio of the PUPD is not better than n on the number of inversions, i.e., error function (1).

For any higher possible maximum rank value $k' > k$, the claimed lower bound on the competitive ratio will automatically apply since the same input sequence S is feasible for k' too. ∎

B. Randomized competitive ratio

The above competitive analysis was specified in terms of a deterministic adversarial packet sequence S. However, assuming an adversary can precisely control the succession of packet ranks as received by a P4 switch is somewhat unrealistic. Below, we turn to a weaker adversary model, where the adversary can choose the probability distribution of the packet ranks $\mathcal{P}$, but the exact succession of packet ranks S is not under control. Assuming that packet rank probabilities $\mathcal{P}$ across the input sequence are *i.i.d.*, the objective is to minimize the expected number of inversions in terms of $\mathcal{P}$ can be expressed as:

$$U_{\texttt{stoch}}(\mathcal{P}) = \mathbf{E}_{\mathcal{P}}\left(U_{\texttt{det}}(S)\right). \tag{2}$$

Theorem 2: Given n queues, $k \geq n+1$ input ranks, and a stationary packet rank probability distribution $\mathcal{P}$, the competitive ratio of PUPD is not better than n, in terms of error function (2).

Proof: Just as in the proof of Thm. 1, first, we will set $k := n+1$. We divide the input rank sequence into *phases*, where phase i starts when the i^{th} rank-1 packet arrives. We construct a packet rank distribution $\mathcal{P}$ such that, in each phase, PUPD makes n inversions *with high probability* (WHP), while with $\underline{q}_{\texttt{offline}} = [1,3,4,5,\ldots,n+1]$, 1 inversion per phase incurs WHP. We note that, in the proof of Thm. 1 (forged for the deterministic setting), S can be divided similarly.

We can observe that the arrival time T_i of the next rank-i packet has a Geo(p_i) distribution, where p_i denotes the probability that the next packet will be a rank-i packet. Thus, $\mathbf{E}(T_i) = {}^1\!/_{p_i}$. Let $p_1 = \epsilon_1$, $p_{n+1} = \epsilon_{n+1}$, and $p_i = \frac{1-\epsilon_1-\epsilon_{n+1}}{n-1}$ for $i \in \{2, \ldots, n\}$. Suppose both $\frac{1/\epsilon_{n+1}}{n^2} > C$ and $\frac{1/\epsilon_1}{1/\epsilon_{n+1}} > C$, for some sufficiently large constant C. With this, a rank $i \in \{2, \ldots, n\}$ packet is much more likely to arrive than a rank-$(n+1)$ one, that is, on its turn, much more likely to arrive than a rank-1 one[2].

After an initial period, when, finally, a monotone decreasing subsequence $(n+1), n \ldots, 2$ can be chosen out of the packets arrived so far, the bounds of the PUPD are set to $\underline{q}_{\texttt{PUPD}} = [2, \ldots, n+1]$. On average, this happens after less than $n^2 + {}^1\!/_{\epsilon_{n+1}}$ packets. The probability of a rank-1 packet arriving in this time frame is very low. Eventually, a rank-1 packet will arrive, marking the start of a *phase*, and causing a push-down (setting $\underline{q}_{\texttt{PUPD}}$ to $[1, \ldots, n]$), and an inversion in $Q[1]$. WHP, this is followed by an inversion in each other queue $Q[i]$ because of enqueuing a newly arriving rank-i packet in it. Once, a rank-$(n+1)$ packet will arrive, setting q_n to $(n+1)$, and initiating a series of push-ups, WHP leading back to $\underline{q}_{\texttt{PUPD}} = [2, \ldots, n+1]$. Then, the arrival of a rank-1 packet will start a new phase. It is easy to see that PUPD commits n inversions in each phase, WHP.

On the other hand, static setting with queue bounds $\underline{q}_{\texttt{offline}} = [1,3,4,5,\ldots,n+1]$ makes at most 1 inversion per phase, that incurs when the rank-1 packet arrives. This completes the proof for $k \geq n+1$.

The lower bounds on competitivity ratios for any possible maximum rank value $k' > n+1$ will automatically apply since the same input sequences as used above are feasible for k' too. ∎

Note that while the *phases* of the input sequence defined in the proof above may be long, inducing low rates of inversions, if the parameters of the sequence are pushed to the limits; concentrating here on a ‘very low’ arrival rate p_1, and a comparably ‘much lower’ p_{n+1}. Setting p_1 and p_{n+1} to less extreme values can push the inversion rates to considerable levels, although such settings might yield a somewhat smaller lower bound on the stochastic competitivity ratio of PUPD. Conducting such a nuanced stochastic competitivity analysis is beyond the scope of this study, and could be part of a follow-up work.

4. Stochastic offline optimum

We have seen that, even in the stochastic model, PUPD can perform n times worse compared to a simple setting with static queue bounds. Below, we will show that the expected cost of inversions defined by (2) for constant queue bounds can be minimized in polynomial time. The first observation is that, given n queues and k ranks, with bounds fixed at $\underline{q} = [q_1 = 1, q_2, q_3 \ldots, q_n])$ extended with an imaginary queue bound $q_{n+1} = k+1$, the expected error is:

$$U^{\texttt{const}}_{\texttt{stoch}}(\mathcal{P}) = \sum_{i=1}^{n} \left(P_i \sum_{q_i \leq a < b < q_{i+1}} \frac{p_b p_a}{P_i^2} \right), \tag{3}$$

where $P_i = \sum_{j=q_i}^{q_{i+1}-1} p_j$ denotes the probability that the next packet will be enqueued to queue i. Eq. (3) holds since the following. In the fraction, we multiply two probabilities: 1) the

[2]More formally, when we say here some event happens WHP, we claim they happen with probability 1 supposing $C \to +\infty$.

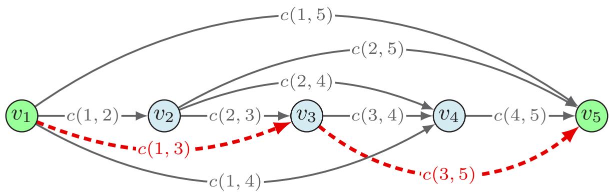


Fig. 2: Example of a DAG representing a possible configuration for $k = 4$. The highlighted path encodes the bounds for a 2-queue SP-PIFO setting ($n = 2$), with the bounds being $q_1 = 1$ and $q_2 = 3$.

probability of the rank of the latest enqueued packet being b is p_b/P_i, and 2) the probability of the next rank in the queue being a (i.e., p_a/P_i). The sum of these values for all the possibly inverted rank pairs in queue i (for b and a s.t. $q_i \leq a < b < q_{i+1}$), when multiplied with the incoming packet intensity P_i and summed over all i, yields (3).

Theorem 3: Let $\mathcal{P}$ the probability distribution of packet ranks and suppose that $\mathcal{P}$ is stationary. Then, the total expected inversion cost in terms of (3) can be minimized in $O(k^2 n)$ time.

Proof: We construct a directed acyclic graph $D(V, A, c)$, where the set of nodes $V = \{v_1, \dots, v_{k+1}\}$ corresponds to the set of possible rank values $\{1, \dots, k\}$ associated with an auxiliary node v_{k+1}, the arc set A stands for arcs (v_i, v_j) : $1 \leq i < j \leq k+1$, and cost $c(v_i, v_j)$ of an arc is the expected error in the queue it represents whenever exactly those packets are enqueued in queue i that have ranks from $\{i, \dots, j-1\}$ (see Fig. 2).[3]

We claim that, for all arcs $(v_i, v_j) \in A$, their costs $c(v_i, v_j)$ (measured in terms of (3)) can be determined in $O(k^2)$ total time. For this, given a fixed lower bound a, we can determine $c(a, a+1)$, $c(a, a+2)$, $\dots$, $c(a, k+1)$, each after, using $O(1)$ basic arithmetic operations per upper bound, where, for $i \geq a+2$, $c(a, i)$ is calculated using the previously calculated cost $c(a, i-1)$ and sum $\sum_{j=a}^{i-1} p_j$. Since there are $O(k^2)$ lower bound– upper bound pairs, the complexity follows. In conclusion, $D(V, A, c)$ can be determined in $O(k^2)$.

Next, we argue that queue bounds set to the node indexes appearing in a shortest $v_1 - v_{k+1}$ path with (at most) n arcs are optimal. Luckily, such a shortest path can be computed in $O(k^2 n)$, e.g., with the help of a slight variation of the Bellman-Ford algorithm [6], see Algorithm 1. The proposition we take advantage of here is that, for any input graph, after i repetitions of the outermost for loop of the Bellman-Ford algorithm, the computed s-t path is a shortest s-t path with at most i edges, if there is an s-t path with at most i edges at all (otherwise it just returns an error and an infinite cost). Applied to our case, a shortest v_1-v_{k+1} path in $D(V, A, c)$ constructed by the Bellman-Ford algorithm in n iterations of the outer for loop suits our needs. Since each iteration takes $O(k^2)$, the runtime complexity of the algorithm follows. ■

We note that the above algorithm works for any conservative cost function c. Thus, the minimization of the expected error

[3]If the expected error is chosed to be measured as defined in (3), then $c(a, b)$ equals $P \sum_{a \leq c < d < b} \frac{p_c p_d}{P^2}$, where $P = \sum_{j=a}^{b-1} p_j$.

Algorithm 1: Modified Bellman-Ford algorithm for the stochastic offline optimum

```
  Input: $D(V, A, c)$
  for $v \in V$ do
1 |   cost$[v] := \infty$; predec$[v] :=$ null
  end
2 cost$[v] := 0$
  for $i = 1..n$ do
      for $(v_a, v_b) \in A$ do
          if $cost[v_a] + c(v_a, v_b) < cost[v_b]$ then
3             cost$[v_b] :=$ cost$[v_a] + c(v_a, v_b)$
4             predec$[v_b] := v_a$
          end
      end
  end
  return $v_1$-$v_{k+1}$ path built from list predec starting at $v_{k+1}$
```

can be done in polynomial time for any cost function c' that is conservative (e.g., non-negative), and polynomially computable. However, the computational complexity for the general case is infeasibly high. To this end, we mention that some related cost formulations meet the convex or concave *Monge* properties [7]. Both cases yield low optimization complexities: [7] shows that finding a cheapest n-link path in a complete DAG with the cost function fulfilling the concave Monge property can be done in $O(k\sqrt{n \log k})$. Then, [8] offers an algorithm that solves the same problem in $k 2^{O(\sqrt{\log n \log \log k})}$, if $n = \Omega(k)$. Note that these complexities do not include determining the necessary cost values.

Finally, in the scope of this paragraph, revisiting the somewhat unrealistic adversary that can precisely control the succession of packet ranks as received by the P4 switch, one can see that with *deterministic* packet sequences, an offline stochastic optimum could be tricked to commit n times more inversions than an offline (deterministic) optimum, e.g., on the following sequence: $S = l \times [2n, 2n-1] + \dots l \times [4, 3] + l \times [2, 1]$ (first packet to arrive on the right, $+$ means concatenation). Note that to lead astray a stochastic offline optimum this heavily, we needed $k = 2n$ ranks, almost twice as many as for PUPD. In fact, a more nuanced claim would be that using $k' \in \{n+1, \dots, 2n\}$ ranks, a stochastic optimum could commit $(k'-n)$ times more inversion than the offline optimum. For this, a possible input sequence S' could be similar to the above S, but some of the neighboring ranks in S had to be mapped together. Thus, in the deterministic setting, a stochastic optimum seems to degrade more gracefully compared to the PUPD in terms of the number of tanks k. Conducting a more nuanced deterministic competitivity analysis for the stochastic optimum is beyond the scope of this study.

5. APPROXIMATING THE OPTIMAL STATIC BOUNDS ONLINE IN CONSTRAINED SPACE

Unfortunately, Algorithm 1 cannot be implemented in real P4 switches. First, an offline algorithm needs the rank distribution $\mathcal{P}$ that is not available in a switch. We solve this problem by learning the rank distribution *online*. Second, P4 switches do not have enough stateful memory to learn the empirical packet rank distribution as this would require $\Theta(k)$ space.

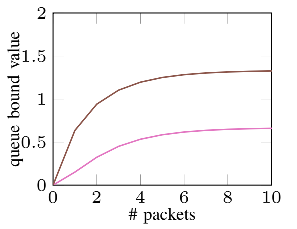
We solve this problem by offering an alternative algorithm that needs only $\Theta(n)$ memory, i.e., the space requirement is proportional to the number of queues, not the number of ranks (which is usually much larger). Of course, shrinking the algorithm's memory footprint results in a loss of optimality. In the evaluations (Sec. 6), we will show that the price we pay for reducing the memory is not prohibitive.

Consider a simplified error function, where the objective is to minimize the maximum per-queue error, instead of the sum of errors:

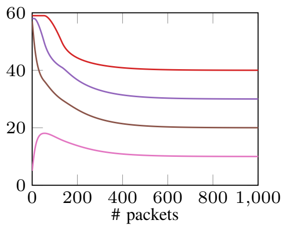
$$U^{\mathtt{max}}_{\mathtt{stoch}}(\mathcal{P}) = \max_{i=1,\dots,n}\left(P_i \sum_{q_i \le a < b < q_{i+1}} \frac{p_b p_a}{P_i^2}\right) . \quad (4)$$

One can observe that minimizing (4) is a special case of the *sequence partitioning problem*, which can be solved in $O(n(k-n))$ time [9]. Also, the space needed for this is reduced to $O(k)$, thanks to the simplicity of the modified objective function (*min-max* instead of *min-sum*).

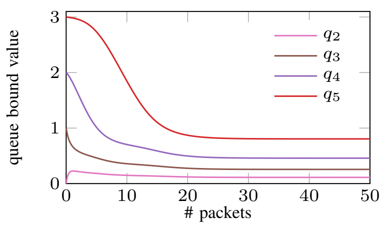
Recall that our aim is to construct an algorithm that fits into $O(n)$ memory. To this end, we need to take care of the space needed to store the learned rank distribution. As a simplification, our objective will be to balance the load on the queues, without caring about the rank distribution inside the rank interval assigned to the queue. In other words, in addition to the queue bounds, we only remember the probability P_i of the next packet being enqueued to queue i. Intuitively, by minimizing the maximum of the P_i values, we even out the load on the queues ($P_i \simeq {}^1/_n$). This will translate to reasonably low inversion rates, measured according to (4), as also reflected in our evaluation results from Sec. 6.



(a) Ranks from $U(0,2)$, queue bounds $\underline{q}_{\text{init}} = [0,0,0]$



(b) Ranks from $U(0,50)$, queue bounds $\underline{q}_{\text{init}} = [0,5,57,58,59]$



(c) Ranks from Exp(2), queue bounds $\underline{q}_{\text{init}} = [0,0,1,2,3]$

Fig. 3: Queue bounds of the continuous relaxation of the Spring over time in case of continuous rank distributions

A. Continuous relaxation

With the above simplifications, our task is now to learn the per-queue loads P_i and, meanwhile, optimize the integer queue bounds to somewhat heuristically optimize (4). To simplify the development, first, we analyse the continuous relaxation of this problem.

In the continuous model, ranks and queue bounds are real-valued. Packets arrive in infinitesimally small quanta, and so per-queue packet rates P_i can be determined for any queue bound setting. We denote the probability density function for the ranks with $f(x)$. Let the two extreme queue bounds be fixed at $q_1 = 0$ and $q_{n+1} = +\infty$. We can express P_i as $P_i = \int_{q_i}^{q_{i+1}} f(x)dx$, for each $i \in \{1,\dots,n\}$. When the optimization reaches a set of stable queue bounds, the following should hold:

$$P_{i-1} = \int_{q_{i-1}}^{q_i} f(x)dx = \int_{q_i}^{q_{i+1}} f(x)dx = P_i \qquad \forall i \in [2,n]$$

For a (small) time quantum Δt, let $\Delta P_i(t)$ denote the number of packets assigned to queue i during time interval $(t-\Delta t, t]$. We define the updated value of of queue bound q_i ($i \in [2,n]$) after this Δt time as:

$$q_i(t) = q_i(t-\Delta t) + \Delta P_i(t) - \Delta P_{i-1}(t). \quad (5)$$

The optimization step, as defined above, just happens to be essentially the same as the Euler method for solving differential equations.

The next Lemma hints that under a variety of circumstances, the optimization as described above converges to the stable equilibrium point, where the loads on the queues are evened out.

Lemma 1: Suppose the following: 1) The probability density function $f(x)$ of the rank distribution is positive on an interval $[0,M]$, for some $M > 0$. 2) On $[0,M]$, the cumulative density function (CDF) of the rank distribution is strictly monotone increasing and continuously differentiable. 3) q_1 and q_{n+1} are fixed at 0 and M, respectively, at all time. 4) At time $t = 0$, we are given some values of the queue bounds $q_1(t) \le q_2(t) \le \dots \le q_{n+1}(t)$. 5) For $i \in \{2,\dots,n\}$, conform with (5), we set $q_i'(t) = (F(q_{i+1}(t)) - 2 \cdot F(q_i(t)) + F(q_{i-1}(t)))$. Then, there exists a globally stable fixpoint of the system, that is $(q_1,\dots,q_{n+1}) = (0, F^{-1}({}^1/_n), F^{-1}({}^2/_n), \dots, F^{-1}({}^n/_n))$. If F is the uniform distribution on $[0,M]$, then $(q_1,\dots,q_{n+1}) = (0, {}^1/_n, \dots, {}^n/_n)$ is *exponentially* stable global fixpoint.
The proof of Lemma 1 is relegated to the Appendix.

Fig. 3 shows our evaluation results of the continuous model over some famous rank distributions. Since we intend to fix q_1 at 0, it is enough to evaluate the rest of the bounds of a system. Fig. 3a shows the trajectory of the relevant 2 queue bounds for a maximal packet rank of 2 accompanied by a uniform rank distribution on $[0,50]$. Further, Fig. 3b shows the relevant queue bounds for the maximal packet rank of 50 and a uniform rank distribution on $[0,50]$. Here, despite the unfortunate initial queue bounds (3 out of the 4 bounds initially exceed the maximum rank), the system quickly converges to the theoretical optimum. Finally, Fig. 3c shows the evolution of queue bounds for an exponential rank distribution.

B. Online learning of the rank distribution

As indicated above, the algorithm, as described so far, should eventually converge to (or slightly oscillate around) a set of stable queue bounds, assuming the packet ranks are i.i.d., but the counters may take on high values as packets are processed. Another problem is, on the other hand, that the number of packets arriving over a short time period Δt is small, yielding imprecise empirical data on the packet rank distribution. Counting the arriving packets with *Exponentially Weighted Moving Averages* (EWMA) solves both problems at once. Let us re-discretize time and the packet arrivals: packets arrive at each positive integer time t (i.e., $\Delta t = 1$), one by one. Let $I_i(t)$ be an indicator (taking on a value of either zero or one) of whether the received packet at time t is assigned to the i-th queue. In addition, let us define a parameter $\alpha \in (0, 1)$ to control how significant new packets should be relative to the packets recorded further in the past (as well as how quickly we forget said packets). We update the EWMA based per-queue packet counters $\mu_i(t)$ on each incoming packet as follows:

$$\mu_i(t) \leftarrow (1 - \alpha) \cdot \mu_i(t - 1) + \alpha \cdot I_i(t) \ ,$$

where $\mu_i(0) \in [0, 1]$ can be set arbitrarily, in Bayesian manner. It is easy to see that $\mu_i \in [0, 1]$ holds at any time for each queue. Furthermore, using the moving averages instead of the ΔP_i values makes the random process of the changing of the queue bounds more stable.

Thus, in this more discrete setting, we rewrite (5) as follows: for all $i \in [2, n]$,

$$q_i(t) = q_i(t - \Delta t) + \mu_i(t) - \mu_{i-1}(t). \qquad (6)$$

Alg. 2 summarizes our heuristic. Since the mechanics of our algorithm resemble the physical model of serially connected springs, we call our algorithm the *Spring* heuristic. In Alg. 2, we still have to re-discretize the queue bounds. Thus, in the algorithm, we keep track of a real-valued version r_i of each queue bound q_i. More precisely, the often minor adjustments of the optimization are done on the continuous r_i bounds (line 8), while the actual queue bounds q_i are the corresponding integer roundings (line 10). This way, the actual queue bounds are just coarse-grained approximations of the underlying fine-resolution representations.

Another issue is that a careless bound adjustment may result in the bound of a queue falling below that of the lower-ranked neighbour during a transient (recall Fig. 3). To avoid this problem, we have implemented an additional *collision detection* mechanism (lines 6-10), which ensures that q_i is never pushed above $q_{i+1} - 1$, or below $q_{i-1} + 1$. This addition guarantees that queue bounds remain integral and sorted in ascending order during the progression of the algorithm.

C. P4 compatibility and resource usage

The Spring heuristic should be well within the capacity of most P4-compatible devices, thus conducting a thorough P4 compatibility analysis of our heuristic was not in the main scope of this study. Regarding the semantics, we note that there are already examples of fixed point arithmetics implemented in P4 [10], and there are also existing P4 implementations of EWMA itself [11]. In terms of memory, the Spring algorithm, as described, should not require considerably more registers than the SP-PIFO itself or the related AIFO [12], which are arguably P4 compatible.

Algorithm 2: Spring heuristic

```
   // Initialization:
1  $[q_1, \ldots, q_n] := [1, \ldots, n]$ // queue bounds
2  $[r_1, \ldots, r_n] := [1, \ldots, n]$ // q. bounds lin. relaxed
3  $[\mu_1, \ldots, \mu_n] := [0, \ldots, 0]$ // error costs
   while Packet arrives with rank $j$ do
4      Packet enqueued into queue $Q[i]$ s.t. $j \geq q_i$, where $i$ is
         the greatest queue index satisfying this condition
5      $[\mu_1, \ldots, \mu_n] := [\mu_1, \ldots, \mu_n] * (1 - \alpha)$ ; $\mu_i := \mu_i + \alpha$
       for $i = n, \ldots, 2$ do
6          lowerBound := $r_{i-1} + 1$ ; upperBound := $+\infty$
7          if $i < n$ then
               upperBound := $r_{i+1} - 1$
           end
8          $r_i := r_i + \mu_i - \mu_{i-1}$
9          $r_i := \min\{\max\{\text{lowerBound}, r_i\}, \text{upperBound}\}$
10         $q_i := \text{round}(r_i)$
       end
   end
```

6. EVALUATION

To make our measurements reproducible and comparable to earlier work, we reused an already existing version of NetBench [13], which contained a reference implementation of SP-PIFO. The simulations used the upstream traffic generators from NetBench, labelled 'Uniform', 'Poisson', 'Exponential', 'Inv. exp.', 'Convex', and 'Minmax', respectively, all of them generating i.i.d. integer packet ranks on interval $[0, 100)$. The Exponential and Poisson generate random numbers from distributions $\text{Exp}(1/25)$ and $\text{Pois}(50)$, respectively, and map them to integer values in $[0, 100)$. The Inv.exp. is based on the Exponential distribution, but it subtracts the generated integer from 99. The Convex distribution is based on a random variable $X \sim \text{Pois}(50)$, with the value of Convex being $(X - 1) \mod 100$ (note that in case of the Convex, rank 99 will be the most probable). Finally, Minmax is based on a variable $Y \sim$ Convex, with a value of $(Y - 10) \mod 50$.

The configuration of the PUPD matches those used in the inversion-related measurements of [5], with a queue number of $n = 8$. The Spring heuristic uses the same parameters as PUPD, with the addition of a parameter $\alpha = 0.01$ to tune the EWMA component. As a benchmark, we also made measurements with the Greedy heuristic of [5] with basic parameters. In the long turn, with i.i.d. ranks, the bounds of Greedy are considered to converge to the optimum, but its space requirement is infeasibly high [5]. The measurements were configured with a one-second limit on the simulated runtime, resulting in around 10^6 packets.

Fig. 4 summarizes the relative performance of the heuristics in terms of the inversion count built into the NetBench implementation of SP-PIFO. We note that this inversion counter may differ from our simplified metrics, and that the number of inversions consistently exceeded 10^5. Despite using the NetBench's own inversion counter, Spring produced a consistently

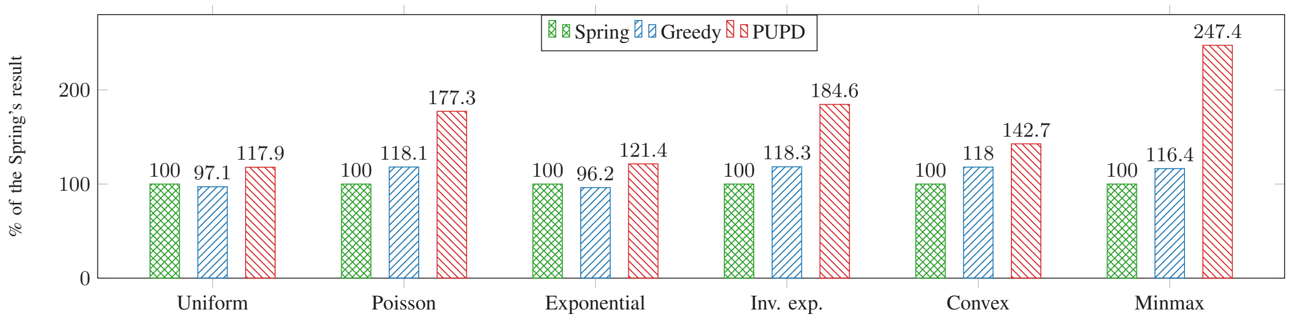


Fig. 4: Number of rank inversions of the different heuristics as percent of the Spring's results

and significantly smaller number of inversions compared to the PUPD, surpassing it on all the distributions studied, with committing only around 40% to 85% of the inversions made by the PUPD. Compared to Spring, PUPD performed the worst at the Minmax, Inv.exp., and Poisson distributions, with roughly 1.7 to 2.5 times more inversions incurred. In the case of the Uniform and Exponential, this ratio was a more moderate 1.2.

Compared to the Greedy, Spring produced a similar number of inversions, beating it on 4 out of the 6 distributions studied. Here, Spring performed the best at the Inv.exp. and Convex distributions committing around 85% of the inversions made by the Greedy, while, at worst, in case of the Uniform and Exponential distributions, this ratio was no more than 1.04.

We have also evaluated the sum of the rank differences of the inverted packet pairs. As Table I demonstrates, for most distributions, the Spring performed slightly better compared to Greedy. The extremes are the Exponential and Inv. exp., where the total inversion size of Greedy is 0.6 and 2.85 times the Spring's, respectively. PUPD is consistently worse, on average making 4.45 times more total inversion size than the Spring.

One intuitive argument on why the PUPD performed consistently the worst among the competing algorithms in our experiment is that its queue bounds are very hectic, driving it to frequent and often unnecessary inversions. See also its behavior on packet sequences built in Sec. 3. Here, a relative advantage of Spring ia its relative reluctance to quickly change the queue bounds. To be fair to PUPD, one of its advantages can be that it reacts relatively effortlessly by definition in case of a quickly changing packet rank distribution. In such case, Spring's reaction time depends on parameter α defining its rate of learning.

7. Discussion

Packet scheduling has been extensively studied for decades [14]–[23]. The concept of programmable scheduling was introduced by [4], [24], which proposed the PIFO queue as an enabling abstraction. While promising, implementing PIFO queues in hardware proved challenging. Hence, some works have suggested new hardware designs such as PIEO [25], BMW-Tree [26], BBQ [27], and Sifter [28]. Others have focused on approximating PIFO behaviors on existing programmable data planes (using and efficiently managing a set of FIFO queues): SP-PIFO [5], QCluster [29], PCQ [30], AIFO [12], Gearbox [31], and PACKS [32]. Spring falls into this latter category. Unfortunately, a thorough formal convergence analysis of these approaches, in most cases, is hard to achieve. While the theoretically provable convergence and convergence speed of the Spring to its equilibrium point has no direct implication to its often more complex counterparts, it suggests similar convergence behaviors for similar frameworks.

TABLE I
Sum of the rank difference of the inverted packets in percent of the Spring's results.

	Uniform	Poisson	Exponential	Inv. Exp.	Convex	Minmax
Spring	100	100	100	100	100	100
Greedy	122	158	60	285	101	107
PUPD	290	462	145	904	167	703

8. Conclusion

Motivated by the industry trends towards rendering the network data plane comprehensibly reconfigurable [3], in this study, we revisited the theory and algorithms for programmable packet scheduling.

We presented the first competitive analysis of heuristic PUPD presented in the SP-PIFO framework for approximating theoretical PIFO queues. We encountered a strong negative result: the PUPD may commit up to n times more packet rank inversions than inevitably needed, with both deterministic and stochastic input, where n is the number of SP queues in the system. In other words, the ability of PUPD to take advantage of an additional SP queue largely decreases, and the algorithm gets further from the optimum as the number of SP queues on disposal grows.

Motivated by this finding, we propose an algorithm to compute the optimal static queue bounds minimizing the expected rate of inversions in case of a given rank distribution. The algorithm runs in polynomial time: its complexity is $O(k^2n)$, where k is the maximum rank.

Considering the online setting, a new online bounds adaptation heuristic called *Spring* was also proposed. Crucially, Spring is easy to reason about formally, which is not the case for PUPD, and it provides favorable results during evaluations: in our measurements, it committed up to 2 times fewer inversions than the PUPD.

ACKNOWLEDGEMENTS

Co-funded by the European Union

The author would like to express his sincere gratitude to Gábor Rétvári and Csaba Sarkadi for their support in this research. This study has received funding from the European Union's Horizon Europe research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101155116. Research partially supported by the National Research, Development and Innovation Fund of Hungary (grant No. 135606).

REFERENCES

[1] B. Vass, C. Sarkadi, and G. Rétvári, "Programmable Packet Scheduling With SP-PIFO: Theory, Algorithms and Evaluation," in *IEEE INFOCOM Workshop on Networking Algorithms (WNA)*, London, United Kingdom, May 2022.

[2] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[3] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Comput. Surv.*, vol. 54, no. 4, 2021.

[4] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 44–57. [Online]. Available: DOI: 10.1145/2934872.2934899

[5] A. G. Alcoz, A. Dietmüller, and L. Vanbever, "SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/alcoz

[6] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[7] A. Agarwal, B. Schieber, and T. Tokuyama, "Finding a minimum weight k-link path in graphs with Monge property and applications," in *Proc. ACM Symposium on Computational Geometry*, 1993, pp. 189–197.

[8] B. Schieber, "Computing a minimum weight k-link path in graphs with the concave Monge property," *Journal of Algorithms*, vol. 29, no. 2, pp. 204–222, 1998.

[9] B. Olstad and F. Manne, "Efficient partitioning of sequences," *IEEE Transactions on Computers*, vol. 44, no. 11, pp. 1322–1326, 1995.

[10] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: DOI: 10.1145/3098822.3098829

[11] A. Fingerhut, P4," "Floating point operations in https://github.com/jafingerhut/p4-guide/blob/d03b4d726a75192f8c7cb7e2ee0d4fceda3bacca/docs/floating-point-operations.md, 2020.

[12] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin, "Programmable packet scheduling with a single queue," ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 179–193. [Online]. Available: DOI: 10.1145/3452296.3472887

[13] G. Memik, W. Mangione-Smith, and W. Hu, "NetBench: a benchmarking suite for network processors," in *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, 2001, pp. 39–42.

[14] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queuing Algorithm," in *ACM SIGCOMM*, New York, NY, USA, 1989.

[15] P. E. McKenney, "Stochastic Fairness Queueing," in *IEEE INFOCOM*, 1990.

[16] D. D. Clark, S. Shenker, and L. Zhang, "Supporting Real-time Applications in an Integrated Services Packet Network: Architecture and Mechanism," in *ACM SIGCOMM*, Baltimore, MD, USA, 1992.

[17] M. Shreedhar and G. Varghese, "Efficient Fair Queueing Using Deficit Round Robin," in *ACM SIGCOMM*, Cambridge, Massachusetts, USA, 1995.

[18] P. Goyal, H. M. Vin, and H. Chen, "Start-time Fair Queueing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," in *ACM SIGCOMM*, Palo Alto, CA, USA, 1996.

[19] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "Pfabric: Minimal near-optimal datacenter transport," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 435–446, Aug. 2013. [Online]. Available: DOI: 10.1145/2534169.2486031

[20] L. E. Schrage and L. W. Miller, "The Queue M/G/1 with the Shortest Remaining Processing Time Discipline," 1966.

[21] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-Agnostic Flow Scheduling for Commodity Data Centers," in *USENIX NSDI*, Oakland, CA, USA, 2015.

[22] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating Fair Queueing on Reconfigurable Switches," in *USENIX NSDI*, Renton, WA, USA, 2018.

[23] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal Packet Scheduling," in *USENIX NSDI*, Santa Clara, CA, USA, 2016.

[24] A. Sivaraman, S. Subramanian, A. Agrawal, S. Chole, S.-T. Chuang, T. Edsall, M. Alizadeh, S. Katti, N. McKeown, and H. Balakrishnan, "Towards Programmable Packet Scheduling," in *ACM HotNets*, Philadelphia, PA, USA, 2015.

[25] V. Shrivastav, "Fast, Scalable, and Programmable Packet Scheduler in Hardware," in *SIGCOMM '19*, Beijing, China, 2019.

[26] R. Yao, Z. Zhang, G. Fang, P. Gao, S. Liu, Y. Fan, Y. Xu, and H. J. Chao, "BMW Tree: Large-scale, High-throughput and Modular PIFO Implementation using Balanced Multi-Way Sorting Tree," in *ACM SIGCOMM*, New York, NY, USA, 2023.

[27] N. Atre, H. Sadok, and J. Sherry, "BBQ: A Fast and Scalable Integer Priority Queue for Hardware Packet Scheduling," in *USENIX NSDI*, Santa Clara, CA, USA, 2024.

[28] P. Gao, A. Dalleggio, J. Liu, C. Peng, Y. Xu, and H. J. Chao, "Sifter: An Inversion-Free and Large-Capacity Programmable Packet Scheduler," in *USENIX NSDI*, Santa Clara, CA, USA, Apr. 2024.

[29] T. Yang, J. Li, Y. Zhao, K. Yang, H. Wang, J. Jiang, Y. Zhang, and N. Zhang, "QCluster: Clustering Packets for Flow Scheduling," 2020.

[30] N. K. Sharma, C. Zhao, M. Liu, P. G. Kannan, C. Kim, A. Krishnamurthy, and A. Sivaraman, "Programmable Calendar Queues for High-speed Packet Scheduling," in *USENIX NSDI*, Santa Clara, CA, USA, 2020.

[31] P. Gao, A. Dalleggio, Y. Xu, and H. J. Chao, "Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing," in *USENIX NSDI*, Renton, WA, USA, 2022.

[32] A. Gran Alcoz, B. Vass, P. Namyar, B. Arzani, G. Rétvári, and L. Vanbever, "Everything matters in programmable packet scheduling," in *22st USENIX Symposium on Networked Systems Design and Implementation (NSDI 2025)*, 2025.

[33] G. Teschl, *Ordinary differential equations and dynamical systems*. American Mathematical Soc., 2012, vol. 140.

[34] "Uber die abgrenzung der eigenwerte einer matrix," *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques*, no. 6, pp. 749–754, 1931.

[35] P. A. F. Parsiad Azimzadeh. Weakly chained matrices, policy iteration, and impulse control. Accessed: 2023. [Online]. Available: *arXiv:1510.03928*

[36] S. Noschese, L. Pasquini, and L. Reichel, "Tridiagonal Toeplitz matrices: properties and novel applications," *Numerical linear algebra with applications*, vol. 20, no. 2, pp. 302–326, 2013.

Balázs Vass received his MSc degree in applied mathematics at ELTE, Budapest in 2016. He finished his PhD in informatics in 2022 on the Budapest University of Technology and Economics (BME). His research interests include networking, survivability, combinatorial optimization, and graph theory. He was an invited speaker of COST RECODIS Training School on Design of Disaster-resilient Communication Networks '19. He has been a TPC member of IEEE INFOCOM since 2023.

APPENDIX

Proof of Lemma 1: For $i \in \{0, \ldots, n\}$, let $r_i = q_{i+1} - F^{-1}(i/n)$. Then, $r_0(t) = r_n(t) = 0$, for every $t \geq 0$, thus they do not make any change from the stability perspective, and can be omitted from further investigation. For $i \in \{1, \ldots, n-1\}$, we have:

$$\begin{aligned} r_i' &= (q_{i+1} - F^{-1}(i/n))' = q_{i+1}' - 0 = \\ &= F(q_{i+2}(t)) - 2 \cdot F(q_{i+1}(t)) + F(q_i(t)). \end{aligned}$$

For shorthand notation, we will use $F_i(t) := F(q_{i+1}(t))$. With this, we define the following matrix in the function of t:

$$A_F(t) = \begin{pmatrix} -2F_1(t) & F_2(t) & 0 & \ldots & 0 \\ F_1(t) & -2F_2(t) & F_3(t) & 0 & \vdots \\ 0 & F_2(t) & -2F_3(t) & F_4(t) & 0 \\ \vdots & \iddots & \ddots & \ddots & \ddots \\ 0 & \ldots & 0 & F_{n-2}(t) & -2F_{n-1}(t) \end{pmatrix} \tag{7}$$

Using $A_F(t)$, we can write:

$$\dot{r}(t) = A_F(t) \cdot r(t). \tag{8}$$

To prove that $r = 0$ is globally stable, we will use the following Proposition.

Proposition 1 (Corollary 3.6 of [33]): The linear system $\dot{r}(t) = A \cdot r(t)$, $r(0) = r_0, A \in \mathbb{R}^{m \times m}$ is asymptotically stable if and only if all eigenvalues α_j of A satisfy $\text{Re}(\alpha_j) < 0$. Moreover, in this case there is a constant $C = C(\alpha)$ for every $\alpha < \min\{-\text{Re}(\alpha_j)\}_{j=1}^{m}$ such that

$$||e^{At}|| \leq Ce^{-t\alpha}, \quad t \geq 0. \tag{9}$$

By Prop. 1, for showing the global stability, it is enough to show that $\text{Re}(\lambda) < 0$ for all eigenvalues of A_F. Gershgorin's circle theorem [34] applied here tells that every eigenvalue of A_F lies in at least one of the disks $D(a_{ii}, \sum_{j \neq i} a_{ji})$. Since, in A_F, $a_{ii} \geq \sum_{j \neq i} a_{ji}$ (i.e., A_F is weakly diagonally dominant, WDD), for each eigenvalue λ we have that either $\text{Re}(\lambda) < 0$, or $\lambda = 0$. This latter case, however, cannot happen since A_F is a *weakly chained diagonally dominant* (WCDD) matrix (defined in a moment), and WCDD matrices are non-singular by [35, Lemma 3.2.]. Here, according to [35, Def. 3.1.], a matrix B is WCDD, if it is WDD, and for each column r, there is a directed path from r to a strictly diagonally dominant column p in the digraph of $D_B = (V_B, A_B)$, where $(i,j) \in A_B$ exactly if $B_{ij} \neq 0$. Clearly, A_F is WCDD.

The global stability of system (8) at $(q_1, \ldots, q_{n+1}) = (0, F^{-1}(1/n), F^{-1}(2/n), \ldots, F^{-1}(n/n))$ follows from the fact that any system yielded by fixing matrix $A_F(t)$ at time point t is globally (exponentially) stable at $(0, F^{-1}(1/n), F^{-1}(2/n), \ldots, F^{-1}(n/n))$.

Finally, the exponential stability of (8) if F is uniform on $[0, M]$ is imminent from Prop. 1 combined with the observation that, in that case, $A_F(t)$ is just a constant tridiagonal matrix A with its elements on the main diagonal equalling -2, while on the upper and lower diagonal, equalling 1. By [36, Eq. (4)], we know that its eigenvalues are $2 \cdot (-1 + \cos \frac{k\pi}{n})$, $k \in \{1, \ldots, n-1\}$. Further investigating the eigenvalues, we see that the largest among them equals $2(\cos \pi/n - 1)$, since the cosine function is monotone decreasing on interval $[0, \pi]$. Thus, in Prop. 1, we can use e.g., $\alpha = 1 - \cos \pi/n$. With this α, combining Eq. (8) and Eq. (9), we have:

$$\begin{aligned} ||r(t)|| &= e^{At} \cdot r(0) \leq ||e^{At}|| \cdot ||r(0)|| \leq \\ &\leq Ce^{-t\alpha} ||r(0)|| = Ce^{t(\cos \pi/n - 1)} ||r(0)||, \end{aligned} \tag{10}$$

for some appropriate constant C. This gives a hint of how fast the queue bounds are converging to the fix point. ■