Programmable Packet Scheduling With SP-PIFO: Theory, Algorithms and Evaluation

Balázs Vass*,†, Csaba Sarkadi*, Gábor Rétvári*

*High Speed Networks Laboratory (HSNLab), Department of Telecommunications and Media Informatics, Faculty of Electrical Engineering and Informatics (VIK), Budapest University of Technology and Economics
† EIT Digital Industrial Doctoral School, {balazs.vass,retvari}@tmit.bme.hu, sarkadicsa@gmail.com

Abstract—Push-In First-Out (PIFO) is a theoretical hardware model for programmable packet scheduling, enabling scheduling policies to be comprehensibly and dynamically reconfigured, and SP-PIFO is a practical emulation for PIFO that can be readily implemented with stock P4 switches. The efficiency of SP-PIFO hinges on a simple heuristic, Push-Up/Push-Down (PUPD), which is responsible for dynamically adapting the mapping of input packets to a fixed set of strict priority queues so as to minimize the rate of scheduling errors with respect to an ideal PIFO. In this paper, we present the first formal analysis of the PUPD algorithm. Our competitive analysis yields that the capacity of PUPD to emulate an optimal PIFO model is getting linearly worse as we keep on adding priority queues to the system. Motivated by this finding, we present an optimal offline scheme, which, given a stochastic model of the input, outputs the optimal SP-PIFO configuration in polynomial time, and we introduce an online heuristic that aims to approximate the offline optimum without requiring a stochastic input model. Our simulations show that the online algorithm can improve the performance of SP-PIFO by a factor of 2× in certain configurations.

I. INTRODUCTION

Traditionally, fixed-function switches implement a specific set of 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 [1]. Packet scheduling in P4 switches, however, 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 [2], [3]. 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 or STFQ [3], 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, [4]). 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 of 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 again Fig.1). 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, and 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.

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 it is known whether more efficient algorithms could be defined to drive SP-PIFO. Our main goal in this study is to fill this gap.

The main contributions are as follows. After a short recap on programmable scheduling (§ II), we present the first competitive analysis of PUPD (§ III). Our results are mostly 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 in accordance with the learned distribution. ‘learns’ the rank distribution and adaptively adjusts the SP-optimal algorithm. Our result in this context is a polynomial-queue bounds in SP-PIFO.

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 in accordance with the learned distribution (§ IV).

Our evaluations (§ V) suggest that the new algorithm can approximate the optimal queue bounds more efficiently than PUPD. We conclude the paper in § VII.

II. 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 \( q_i+1 \) 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 behavior, 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 [4].

The efficiency of SP-PIFO to emulate an optimal PIFO model is contingent on its capacity to consistently map low-rank packets to high-priority queues, 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.

III. 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 below results 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:

\[
U_{\text{det}}(S) := \#\{(a,b) \subseteq S : a < b \}
\]

Our next result shows that, for a certain sequence of packets \( S \), PUPD produces \( n \) times more inversions than the optimal offline algorithm in terms of the error function \( U_{\text{det}}(S) \), where \( n \) is the number of SP queues. Intuitively, PUPD is getting further from the optimum as we add more 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).

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

Proof: First, let \( k := n + 1 \). Let the input rank sequence be as follows (first packet to arrive on the right):

\[
S = l × [n, n, \ldots, 2, 1, 2, \ldots, n, n+1]
\]

We theorem the queues built by the PUPD are (first packet enqueued to and dequeued from right):

\[
Q_{\text{PUPD}}[i] = [i, i+1] × l, \quad \forall i \in \{1, \ldots, n\}.
\]

This means a number of \( l \) inversions per queue, that sums up to \( nl \) for the PUPD.

On the other hand, by setting constant offline queue bounds to \( q = [q_1 = 2, q_2 = 3, \ldots, q_n = n+1] \), the queues built are:

\[
Q_{\text{offline}}[1] = [2, 1, 2] × l,
\]

\[
Q_{\text{offline}}[i] = [i + 1] × (2l), \quad \forall i \in \{2, \ldots, n-1\},
\]

\[
Q_{\text{offline}}[n] = [n + 1] × (l).
\]
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 \( l \geq 1 \). This means that the competitive ratio of the PUPD is at least \( n \) on the number of inversions, i.e., error function (1).

\[ U_{\text{stoch}}(P) = \mathbb{E}_P(U_{\text{det}}(S)). \]  

**Theorem 2:** Given \( n \) queues and \( k = n + 1 \) input ranks and a stationary probability distribution \( P \) of packet ranks, the competitive ratio of PUPD is at least \( n \) in terms of error function (2).

**Proof:** Let us divide the input rank sequence into phases, where phase \( i \) starts when the \( i^{\text{th}} \) rank-1 packet arrives. We will construct a packet rank distribution \( P_i \), for which, in each phase, PUPD makes \( n \) inversions with high probability (WHP), while with \( q_{\text{off-lining}} = [1, 3, 4, 5, \ldots, n + 1] \), only 1 inversion per phase is incurring WHP. We note that \( S \) in the proof of Thm. 1 can be divided similarly.

Observe that, since the arrival time \( T_i \) of the next rank-1 packet has a Geo(\( \mathcal{P}(i) \)) distribution, \( \mathbb{E}(T_i) = 1/\mathcal{P}(i) \). Let \( \mathcal{P}(1) = \epsilon_1, \mathcal{P}(n + 1) = \epsilon_{n+1}, \) and \( \mathcal{P}(i) = \frac{1-\epsilon_i-\epsilon_{i+1}}{n+1} \) for \( i \in \{2, \ldots, n\} \), and suppose \( n^2 \ll 1/\epsilon_{i+1} \ll 1/\epsilon_i \). With this, a rank \( i \in \{2, \ldots, n\} \) packet is much more likely to arrive than a rank-(\( n + 1 \)) one, that is much more likely than a rank-1 one.

We claim that after an initial learning period (when 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 \( q_{\text{PUPD}} = [2, \ldots, n + 1] \). On average, this happens after less than \( n^2 + 1/\epsilon_{i+1} \) packets. The probability of a rank-1 packet arriving during this time is very low.

Eventually, a rank-1 packet arrives, starting a phase, and causing a push-down (setting \( q_{\text{PUPD}} \) 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 some rank-1 packets in them. Once, a rank-(\( n + 1 \)) packet arrives, setting \( q_{n} \) to \( (n + 1) \), and initiating a series of push-ups, WHP leading back to \( q_{\text{PUPD}} = [2, \ldots, n + 1] \). Then, the arrival of a rank-1 packet starts a new phase. It is easy to see that PUPD commits \( n \) inversions per phase, WHP.

Clearly, a static setting with queue bounds \( q_{\text{off-lining}} = [1, 3, 4, 5, \ldots, n + 1] \) makes 1 inversion per phase WHP (that incurs when the rank-1 packet arrives). The proof follows. ■

**B. Randomized competitive ratio**

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

\[ U_{\text{stoch}}(P) = \mathbb{E}_P(U_{\det}(S)). \]  

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

**Proof:** First we construct a directed acyclic graph \( D(V, A, c) \), where the set of nodes \( V = \{v_1, \ldots, v_{k+1}\} \) corresponds to the set of possible rank values \( \{1, \ldots, k\} \) associated with a helper node \( v_{k+1} \), the arc set \( A \) stands of arcs \( \{v_i, v_j\} : 1 \leq i \leq j \leq k \) and cost \( c(v_i, v_j) \) of an arc is the expected error in queue \( i \) whenever packets with ranks \( \{i, \ldots, j - 1\} \) are queued in queue \( i \) (see Fig. 2).

We claim that, for all arcs \( \{v_i, v_j\} \in A \), their costs \( c(v_i, v_j) \) can be determined in \( O(k^2) \) total time as follows. For a fixed lower bound \( a \), we can determine \( c(a, a + 1), c(a, a + 2), \ldots, c(a, k + 1) \), 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 \( c(a, i - 1) \). Since there are \( O(k^2) \) lower bound– upper bound pairs, the complexity follows. Based on this, \( D(V, A, c) \) can be determined in \( O(k^2) \).

Next, we show 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. Such a shortest path can be computed in \( O(k^2n) \), e.g., with the help of a slight variation of the Bellman-Ford algorithm [5], see Algorithm 1. The proposition we take
advantage of is that, in any input graph, if there is an \( s \)-\( t \) path with at most \( i \) edges, then, after \( i \) repetitions of the outermost loop for the Bellman-Ford algorithm, the computed \( s \)-\( t \) path is a shortest \( s \)-\( t \) path with at most \( i \) edges. Thus, 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 \( O(k^2) \) time, the complexity of the algorithm follows.

We note that the above algorithm works for any conservative cost function \( c \). Thus, as long as an alternative non-negative cost \( c' \) is polynomially computable, the minimization of the expected error can be done in polynomial time.

Also, some related cost formulations meet the convex or concave Monge properties [6]. Both cases yield low optimization complexities: [6] 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 \( O(k^2 \sqrt{n} \log k) \). Then, [7] 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.

V. APPROXIMATING THE OPTIMAL STATIC BOUNDS ONLINE IN CONSTRAINED SPACE

Unfortunately, Algorithm 1 cannot be implemented in real P4 switches. First, the offline algorithm needs the rank distribution \( P \) 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; this would require \( \Theta(k) \) space. We solve this problem by showing an 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, this will result in a loss of optimality; in § VI we show that the price we pay for reducing the algorithm’s memory footprint 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:

\[
(T^\text{max})_{\text{stoch}}(P) = \max_{i=1, \ldots, n} \left( P_i \sum_{q_i \leq a < q_i + 1} \frac{p_b p_a}{P_i^2} \right).
\]

It is easy to see that minimizing (4) is a special case of the sequence partitioning problem, which can be solved in \( O(n(k - n)) \) time [8]. 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, we want 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. 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 (\( P_i \approx \frac{1}{n} \)). As per (4), this will translate to a reasonably low inversion rate.

A. Continuous relaxation

Our task is now to learn the per-queue loads \( P_i \), and meanwhile optimize the integer queue bounds so as to minimize (4). To simplify the development, first, we take a look at the continuous relaxation of this problem.

In our 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. Let \( f(x) \) denote the probability density function for the ranks and let the two extreme queue bounds be fixed at \( q_1 = 0 \) and \( q_{n+1} = +\infty \). Let \( P_i := \int_{q_i}^{q_{i+1}} f(x) dx \) for each \( i \in \{1, \ldots, n\} \). When the optimization reaches a set of stable queue bounds, the following should hold:

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

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

\[
q_i(t) = q_i(t - \Delta t) + \Delta P_i(t) - \Delta P_{i-1}(t).
\]

The optimization step as defined above just happens to be essentially the same as the Euler method for solving differential equations. Fig. 3 shows our evaluation results of the continuous model over some famous rank distributions. Since we intend to always have \( q_1 = 0 \), it is enough to evaluate the rest of the bounds of a system. Fig. 3a shows the trajectory

![Fig. 3: Queue bounds of the continuous relaxation of the Spring over time in case of continuous rank distributions](image-url)
of the relevant queue bounds for the maximal packet rank 50 and a uniform rank distribution on $[0, 50]$. Here, despite the unfortunate initial parameters (3 out of the 4 bounds initially exceed the maximum rank), the system quickly converges to the theoretical optimum. Fig. 3b shows the evolution of queue bounds for an exponential rank distribution.

B. Online learning of the rank distribution

As shown above, the algorithm as described so far should eventually converge 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 that the number of packets arriving over a short time period $\Delta t$ is very small, yielding an imprecise empirical data on the packet rank distribution.

Counting 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 $t$ (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 a new packet should be relative to the packets recorded in the past (as well as how quickly we forget said packets), and 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 $\Delta P_i$ values makes the random process of the changing of the queue bounds more stable.

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

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

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.

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 a coarse-grained approximation of an underlying fine-resolution representation.

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

```
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
5 the greatest queue index satisfying this condition
6 for i = n, \ldots, 2 do
7 lowerBound := r_{i-1} + 1; upperBound := +\infty
8 \text{if } t < n \text{ then}
9 upperBound := r_{i+1} - 1
10 end
11 ri := min{max{lowerBound, r_i}, upperBound}
12 qi := round(r_i)

end

C. P4 compatibility and resource usage

Due to the lack of space, we omit a thorough P4 compatibility analysis of the Spring heuristic. Regarding the semantics, we note that there are already examples of fixed point arithmetics implemented in P4 [9], and there are also existing P4 implementations of EWMA itself [10]. In terms of memory, the Spring algorithm as described should not require considerably more registers than the SP-PIFO itself or the related AIFO [11] which should already be well within the capacity of most P4 compatible devices.

VI. Evaluation

To perform our measurements reproducible and comparable to earlier work, we reused an already existing version of NetBench [12] that contained a reference implementation of SP-PIFO. The simulations used the upstream traffic generators from NetBench, labeled ‘Uniform’, ‘Poisson’, ‘Exponential’, ‘Inv. exp.’, ‘Convex’, and ‘Minmax’, respectively, all of them generating i.i.d. integer packet ranks on the $[0, 100]$ interval. The Exponential and Poisson generate random numbers from distributions Exp($\frac{1}{25}$) and 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 $X \sim \text{Pois}(50)$, where the value of Convex is $(X - 1) \mod 100$. Finally, Minmax is based on a $Y \sim \text{Convex}$, with a value of $(Y - 10) \mod 50$.

The configuration of the PUPD matches those used in the inversion-related measurements of [4]: $n = 8$. The Spring heuristic uses the same parameter as PUPD, with the addition of a parameter $\alpha = 0.01$ to tune the EWMA component. As a benchmark, we also made measurements with heuristic Greedy of [4] with basic parameters. On 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 [4]. The measurements were configured with a one-second limit on the simulated runtime, resulting in around one million packets.
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**

This research was partially supported by the National Research, Development and Innovation Fund of Hungary (grant No. 135606). Supported by the UNKP-21-4 New National Excellence Program of the Ministry of Innovation and Technology from the source of the National Research, Development and Innovation Fund.

**REFERENCES**


