Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Programmable Packet Scheduling: Theory, Algorithms and Evaluation

Scientific Students' Association Report

Author:

Csaba Sarkadi

Advisors:

Dr. Gábor Rétvári
Balázs Vass

2021

Contents

Chapter 1

Introduction

Traditionally, networks were built upon fixed-function switches that implement all their behaviour in hardware. More recently, P4-compatible, programmable switches have emerged on the market, allowing network operators to refine most of the aspects of their data plane, such as parsing and handling new protocols or custom forwarding policies that act upon arbitrary fields of a packet header.

Yet even with programmable switches, scheduling algorithms are still mostly implemented in hardware with fixed functionality. This severely limits the applicability of programmable switches to the few use cases that can be supported with the limited set of built-in fixed-function packet scheduling algorithms, such as FIFOs or priority queuing. Recently, a hardware-assisted abstraction, PIFO [10] was proposed as a generic building block that would allow users to implement a variety of scheduling algorithms at line rate, but no actual hardware implementation exists at the time of writing. This, combined with perceived limitations of the reference PIFO hardware design, led to the appearance of SP-PIFO in 2019[1], a proposed programmable packet scheduler that attempts to approximate PIFO behaviour on existing hardware, using the conventional queuing primitives such as FIFOs and priority queuing available in all switches, combined with a bound adaptation function that maps packet ranks to the aforementioned queues with the goal of minimizing the number inversions among them on the output of the SP-PIFO queue.

In this study, my goal is to provide an independent analysis on the optimality of the PUPD adaptation algorithm proposed by the SP-PIFO authors in order to understand how it fares against static bounds on arbitrary rank-distributions, and whether it is possible to improve upon it. To this end, I have modified NetBench, the simulation framework used in the original SP-PIFO paper[1], both to enable parallel processing of simulations and to make it easier to analyze arbitrary adaptation functions with it. Also proposed is an algorithm for efficiently computing optimal static bounds for a given rank distribution with regards to a given error function, as well as a new heuristic that is capable of adapting queue bounds to unknown packet rank distributions. Extensive evaluations conducted with NetBench show that my heuristic, under mild assumptions, may significantly outperform PUPD on the same input.

This proposed algorithm is also evaluated against the following criteria:

- **Stability:** Does the algorithm converge?
- **Rate of convergence:** If so, how many iterations does it take to converge?
- **Total inversion count:** How many inversions happen under the algorithm?
- **Total inversion cost:** A metric of how closely an algorithm approximates actual PIFO-like behaviour, the lower the better.
- **Complexity:** Simpler algorithms are easier to implement and reason about.
- **P4 compatibility:** Closely related to simplicity, the adaptability of an algorithm to the, in some ways restricted semantics of the P4 language and those of P4-compatible devices.

The rest of this report is organized as follows. In Chapter 2, we briefly review background of packet scheduling in programmable switches, including the PIFO and SP-PIFO proposals we build upon. Next, in Chapter 3, we introduce various error metrics that may be used to quantify the behaviour of an SP-PIFO system, and derive both an algorithm for solving the problem for the case of static queue bounds, as well as a new adaptation algorithm based one of the described error metrics. In Chapter 4, we describe our use of NetBench and proceed to evaluate the simulation results in Chapter 5. Finally, Chapter 6 summarizes our results and the remaining questions to be resolved as part of future work.

Chapter 2

Background

2.1 Packet scheduling

Packet scheduling algorithms arbitrate access between the different traffic flows in a network device to shared communication media [11]. A scheduler typically manages one or more packet queues and, in each time slot, chooses one packet from a queue to be dropped or transmitted to the next-hop device. The decision is made in order to realize various scheduling policies: controlling the rate at which flows can send packets (shaping or policing), adjusting the quality of transmission service provided to individual packet flows (scheduling) providing access to the shared medium without causing starvation to any of the users (fairness), etc.

2.2 Packet scheduling in fixed-function switches

Network switches and routers, as well as endpoint operating systems typically include a fixed set of queuing algorithms baked deeply into the hardware and software. For instance, an off-the-shelf Cisco switch contains an implementation for First-In-First-Out queuing, Priority Queuing, Weighted Fair Queuing, and various combinations of these [5]. Defining a finite set of supported queuing algorithms allows vendors to fine-tune their implementations for maximum performance or minimal resource footprint, but still allows users and operators to build their own QoS, traffic shaping, and fair queuing policies on top of these simple primitives. Unfortunately, this limited flexibility is not sufficient in many emerging applications. As an example, in data centre switching, the objective is often low tail latency or minimal flow completion time [2]; unfortunately, typical switch hardware does not come equipped with the necessary queuing algorithms to support such policies. Another critical use case is real-time applications [6] and time-sensitive networking (TSN, [9]): this application domain specifies its own collection of elaborate scheduling policies, which is scarcely supported in today's fixed-function switches (if at all).

2.3 Push-In First-Out queues

PIFO queues [10] are a proposed building block for scheduling algorithms that rely on priority queues to reorder packets according to their *ranks* (of which there are exactly k, ranging from 0 to $k-1$), numerical values assigned to them by user-defined functions. Whenever a packet arrives, a rank is assigned to it, after which it is pushed into a priority queue. Packets may enter at arbitrary positions according to their rank, but they are always dequeued from the head of the priority queue. These queues can be used to implement both work-conserving algorithms, in which case ranks may be thought of as packet priorities, as well as non-work-conserving algorithms where ranks denote the time at which a given packet should continue to be processed.

While this model does have some limitations (inability to recalculate ranks of already enqueued packets on the arrival of new packets, and only being able to perform traffic shaping on the input of a PIFO), it still allows users of this abstraction to implement a variety of scheduling algorithms without sacrificing performance: because PIFOs are a hardware-assisted abstraction, they have no problems running at line rate.

2.4 Strict-Priority Push-In First-Out queues

Unfortunately, though the proposal was accompanied by a reference hardware implementation, at the time of writing, no commercially available hardware provides PIFO queues. This, combined with the limitations of the reference PIFO hardware design, spurred research into different means of achieving PIFO-like behaviour.

One such alternative approach is SP-PIFO [1], which attempts to approximate the behaviour of a single PIFO queue by using a fixed number of n FIFOs.

Under SP-PIFO, each such FIFO is assigned a *queue bound*. As packets arrive, the FIFOs are enumerated in ascending order of priority, and incoming packets are enqueued into the first queue where the associated queue bound is lower than or equal to the rank of the packet. Queue bounds are dynamically reassigned by an *adaptation function* each time a packet is enqueued. The goal of the adaptation function is to map the incoming packets to the queues in a way that results in the least number of *inversions*, occurrences of a higher-ranked (lower priority) packet appearing on the output of the SP-PIFO queue instead of a lower-ranked packet that is already in the queue, as such events would never happen in a real PIFO implementation (see Fig. 2.1 for an example).

2.4.1 PUPD

One such adaptation function, also introduced in [1] alongside SP-PIFO, is PUPD (*Push-Up Push-Down*).

2.4.1.1 Push-Up

Whenever a packet with rank r is enqueued into Q_i, the bound of that queue (q_i) shall be set to r. Paired with the SP-PIFO scheduling algorithm, this means that whenever a packet is enqueued somewhere, that queue shall have the highest rank enqueued since the last push-down as its queue bounds. This also means that as long as no push down

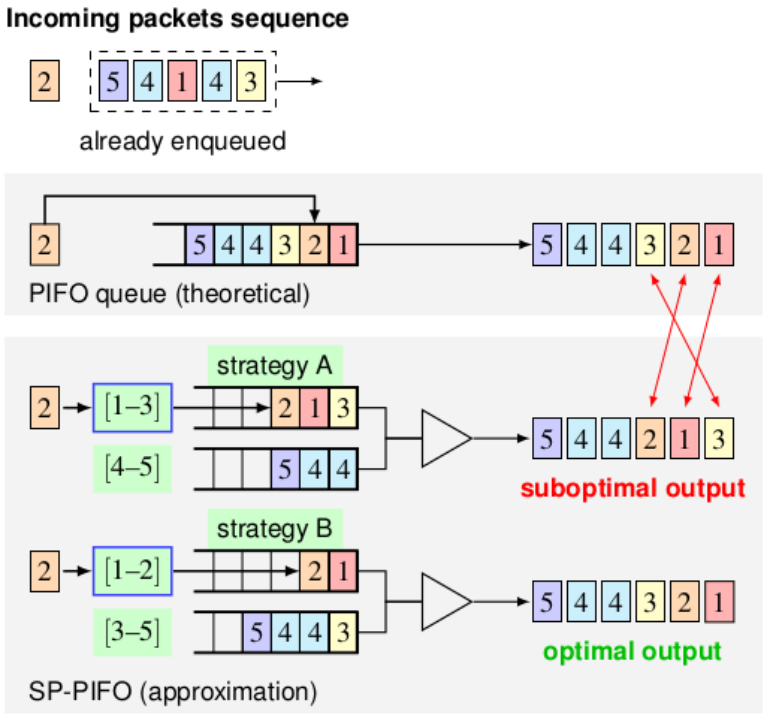


Figure 2.1: SP-PIFO approximates the behavior of PIFO queues by adapting how packet ranks are mapped to priority queues. [1]

happens, packets avoid at least one possible inversion for each lower priority queue they are not enqueued into.

Packets with the same rank as the queue bound are enqueued to the same queue, but since their rank is no lower than the previous highest rank already present, they are not supposed to suffer inversions there. In any of the higher ranked queues, were they enqueued, they would have had suffered an inversion against the known, already enqueued higher-ranked packet.

2.4.1.2 Push-Down

Without a force to balance it, Push-up would naturally cause queue bounds to approach the topmost n ranks. This raises both a problem with regards to what should happen with packets whose rank is lower than the current lowest queue bound, as well as that of eventually ending up with a configuration that does not quite look optimal, that also does not change anymore.

Thus there is a need for another strategy that decreases the queue bounds from time to time. To this end, when a packet with rank $r < q_1$ arrives, the *Push-Down* strategy decreases all queue bounds by the difference between the rank of the enqueued packet and the bound of the highest priority queue, i.e., $q_i := q_i - (q_1 - r)$, for all $i \in \{1, \ldots, n\}$. Note that q_1 is set to r.

Push-downs result in inversions in the highest priority queue, and because such events adjust all queue bounds, they are also linked to all inversions that occur between lower priority queues.

2.4.2 The Greedy Gradient heuristic

Also introduced by [1] is a greedy bounds adaptation algorithm that tracks the rank distribution of incoming packets exactly and attempts to minimize the expected cost of inversions in an SP-PIFO queue based on that information. Without delving too much into the details, as there is a gentle introduction to the problem space later on (Chapter 3), let us re-introduce the problem the Greedy algorithm is based on. With Q_i referring to the i^{th} FIFO in the SP-PIFO queue, which has queue bound q_i, p_x denoting the probability of a packet with rank x arriving into that queue, and finally, $\text{cost}(a, b)$ being some function that tells us how much of a problem we consider it is when a higher-ranked packet b precedes a on the output of the SP-PIFO queue, the goal is to find a set of queue bounds that minimizes $\sum_{i=0}^{n-1} U(Q_i)$ for a given cost function:

$$U(Q_i) = \sum_{r=q_i}^{q_{i+1}-1} \sum_{r'=r+1}^{q_{i+1}-1} p(r)p(r') * \text{cost}(r', r) \ .$$

While this is the underlying problem, the proposed algorithm that seeks to approximate its solution as packets arrive works with a slightly different error estimation. In this case, the cost is estimated not on a per-queue but on a per-packet basis.

Given a sequence of packets P, inserted into queue i, the estimated error (or *unpifoness*) is:

$$U(P, Q_i) = \sum_{p \in P} \text{cost}(p, p'),$$

where p' is the highest rank currently waiting in Q_i. Whenever a packet arrives, the greedy algorithm updates its internal state and periodically (with a configurable step size) estimates the unpifoness of the current queue bounds, as well as how the unpifoness would change for a queue if its bound were nudged one step higher or lower. If any of the changes are estimated to be beneficial, they are taken by this algorithm, hence the greedy qualifier.

Chapter 3

SP-PIFO: modeling and algorithms

3.1 Estimating cost of inversions in SP-PIFO

Under PIFO, which SP-PIFO seeks to approximate, no inversions can happen. Under SP-PIFO, one can think of inversions with regards to both frequency (how many inversions happen over time, at each rank or overall?), as well as the cost of inversions: if two packets with ranks a and b cause an inversion, and $a < b$, then b would have caused an inversion against any packet with rank $a < c < b$ as well, and dequeuing b that much too early is more of a potential problem than dequeuing a packet with rank c would be. Thus, intuitively, for an individual inversion between ranks a and b, the cost of that inversion is proportional to $b - a$.

While [1] did provide a good (but at the same time, not compatible with target hardware) online heuristic against which PUPD can be compared, fixed queue-bounds, which are easy to reason about formally, were only used for evaluating inversions in the case where the ranks of incoming packets have a uniform distribution.

As these bounds seemed to work just as well as the greedy heuristics did for that specific distribution, this raises the question of just how close to optimal these greedy and PUPD algorithms actually are for rank distributions beyond uniform ranks and whether there is any room for improvement.

To examine whether or not static bounds could perform on par with the previously introduced online algorithms, we first need a way to determine the optimal static bounds, and this is a good opportunity to talk about **why** the evenly distributed queue bounds, as used in [1], may be considered optimal for a uniform rank distribution.

3.1.1 Estimating the cost of inversion for independently and uniformly distributed packet ranks

When the distribution of ranks on the input is uniform, we can safely ignore the probabilities of the ranks on our input (as they are the same for any two ranks that may cause inversions) and focus only on minimizing the cost of inversions.

We can either attempt to minimize the total cost of all possible inversions in any of the queues, or the cost of the greatest possible inversions. Since we are, at least for the moment, ignoring probabilities of a given rank appearing on the input, the output of both

functions only depends on the *width* (the difference between q_i and q_{i+1}) of the queue, as specified in Fig. 3.1.

$$U_{\text{sum}}(Q_i) = \sum_{j=q_i}^{q_{i+1}-1} \sum_{k=j+1}^{q_{i+1}-1} (k-j)$$

(a) Estimate based on total cost of all possible inversions in a queue.

$$U_{\text{upperbound}}(Q_i) = \frac{q_{i+1} - 1 - q_i}{2}$$

(b) Estimate based on the greatest possible inversion in a queue.

Figure 3.1: Approaches to estimating errors in a single queue.

In the upper estimation, we divide by two because this greatest possible inversion that may, even with adversarial input, only on every second packet, and we just so happen to have ended up with a function that yields the magnitude of the average inversion in the queue.

By reusing the per-queue cost functions, one can estimate either the total or the greatest per-queue cost of inversions for an entire SP-PIFO setup as given in Fig. 3.2.

$$U(Q) = \sum_{i=0}^{n} U(Q_i)$$

(a) Estimation on the total cost of inversions for a given configuration of queue bounds.

$$U(Q) = \max_{i \in \{0,\ldots,n-1\}} U(Q_i)$$

(b) Estimation on the highest cost of inversions in any queue for a given configuration of queue bounds.

Figure 3.2: Some ways to define the total estimate on the cost of inversion for an SP-PIFO queue, given a per-queue cost function.

With that in mind, minimizing either of the above cost functions then becomes an exercise of minimizing the maximum queue width. Dividing the ranks evenly with queue bounds fits that criteria, and it is not hard to see that any change to this setup would necessarily increase the width of at least one queue, yielding a less favourable configuration.

3.1.2 Estimating the cost of inversions for independently and identically distributed ranks

The uniform distribution is a simple introduction to the underlying optimization problem, but there would not be much use for an online adaptation function if we could just always assume packets were as evenly distributed as that. The various ranks have different probabilities of appearing on the input, and that changes the optimal bounds as well.

Intuitively, if certain ranks appear more often than others, they have more chances to cause or suffer an inversion, and so the total cost of inversions related to these packets becomes greater, even if the costs of individual inversions do not change. In the same vein, if a queue receives more packets than its counterparts, there is a higher probability that packets in that queue participate in an inversion, at least long as there are at least two ranks mapped to that queue.

Assuming p_x stands for the probability of a packet with rank x received by SP-PIFO, and denoted by $P_i = \sum_{j=q_i}^{q_{i+1}-1} p_j$ is the probability that a packet is scheduled onto queue i, we can adjust the error estimations introduced earlier as specified in Fig. 3.3.

$$U_{\text{sum}}(Q_i) = \sum_{j=q_i}^{q_{i+1}-1} \sum_{k=j+1}^{q_{i+1}-1} (k-j)\frac{p_k p_j}{P_i^2} \qquad U_{\text{upperbound}}(Q_i) = \frac{q_{i+1}-1-q_i}{2} P_i$$

(a) Estimated total cost of all possible inversions in a queue, weighted by the probability of packets with ranks j and k following each other in the queue.

(b) Estimate based on the greatest possible inversion, weighted by the probability of packets being scheduled onto the queue in question.

Figure 3.3: Per-queue estimations of inversion costs, accounting for the input rank distribution.

Equation 3.3a in particular should be familiar from the introduction of the Greedy algorithm, but this time it is accompanied by a train of thought leading to it.

Equation 3.3b represents an alternative approach that favours minimizing the worst possible inversion in any of the queues instead of minimizing the total expected inversion cost. It should be noted that it only depends on the widths of each of the queues (this information is already present in the form of queue bounds) as well as the probabilities that a packet ends up in a particular queue. Tracking this probability can easily be done in P4, in the form of a per-queue packet counter, where the number of packets received by each queue, divided by the total number of packets, could give an exact value. Alternatively, since we do not need to limit the estimation to any range, we can just treat the counters themselves as relative weights, which could save us some complexity in a P4 implementation if we sought to build an adaptation algorithm based on this error estimation.

3.2 Computing optimal static queue bounds in case of known rank distribution

Using the error functions introduced above, we are now in a position to compute the *optimal* set of queue bounds for a given rank distribution. In this context, a set of queue bounds is optimal if, for a given stationary rank distribution, it minimizes (one of) the error functions as per Fig. 3.3. Substituting q_i with a and q_{i+1} with b, taking a and b as parameters, we can redefine the functions above to have an interface of $U(b, a)$ to estimate the cost of inversions for a given configuration where all ranks $a \leq r < b$ end up in the same queue.

Assuming we know the number of ranks, we can construct a DAG where nodes denote ranks, and for each pair of nodes labelled a and b (where $b > a$), there is exactly one edge directed from a to b, with $U(b, a)$ serving as a function for the length of the arc. On such a DAG (as depicted in Figure 3.4), we may observe that the nodes along any path from rank zero to (but not including) rank k encode the bounds for a valid configuration of the queue bounds, with the number of queues being equal to the number of edges taken.

Interpreting the estimation for the cost of inversions as link lengths, we can also see that the total length of edges along such a path from zero to k are analogous to the summation

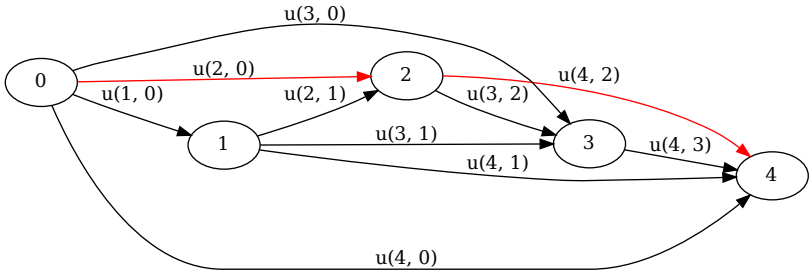


Figure 3.4: Example of a DAG representing the possible configurations for $k = 4$. The highlighted path denotes the bounds for a 2-queue SP-PIFO construction, with the bounds themselves being 0 and 2.

we performed earlier to obtain an estimate on the total cost of inversions for a given configuration of the queue bounds in Equation 3.2a.

As such, finding a shortest path from node zero to node k that consist of exactly n steps should yield us a lowest-cost configuration for our queue bounds. Such a path can be found by slightly modifying a single-source shortest path algorithm.

By replacing addition with $\max(a, b)$ while evaluating total path lengths, it is also possible to use this strategy to solve for cost functions that estimate the maximum inversion cost per queue as shown in Equation 3.2b.

For our purposes, we opted for a variation of the Bellman-Ford algorithm [4] to compute the optimal queue bounds. The key changes were stopping the outermost loop (that performs the relaxation steps) after n iterations, and tracking the preceding nodes for each iteration of this loop separately, required to recover the actual paths of length n. Without the latter change, the algorithm would keep overwriting the length of the path to intermediate nodes, which is indeed needed for the next iteration of the relaxation step to work properly in Bellman-Ford, but in our case, it would preclude us from recovering the lowest cost path towards such an intermediate node itself as it would be necessarily overwritten in the next iteration with a zero cost path.

The reason for that is that an arc going from node x to node $x + 1$ encodes a queue that only receives packets from a single rank x on its input. Naturally, no inversions can happen in such a queue, thus such arcs have a zero length. After each iteration, all intermediate nodes that are reachable along a path consisting only of such zero-cost arcs, would be deemed by the algorithm to be reachable along a zero cost, and this distorted distance tracking propagates to higher-ranked nodes as well, causing the now zero cost nodes to be considered the preceding nodes even if that would ultimately yield a path with more than n edges.

Tracking the preceding nodes separately for each iteration avoids the issue: by always retrieving the preceding node from the saved state of the previous step, it is possible to recover a shortest path that consists of no more than n edges. It is easy to show that, with these modifications, the algorithm terminates in $O(nk^2)$ steps with the optimal queue bounds. The algorithm implements both the per-queue cost estimation methods defined in Figure 3.3 as well as both problems depicted in Figure 3.2. The output of this model has been used in all of NetBench simulations involving static queue bounds[1].

[1] https://github.com/enoperm/offline-model

3.3 Approximating the optimal static bounds online in constrained space

Unfortunately, implementing the optimal algorithm in a P4-switch would be impractical. The major problem is that we do not know the rank distribution; not just that the distribution is not revealed to the switch offline, but it may also rapidly change in time due to sudden shifts in traffic patterns and switch load. Consequently, we need an online algorithm that adaptively "learns" the rank statistics and adjusts the queue bounds dynamically with respect to the learned model.

However, even the online tracking of the rank distribution is a major challenge due to the limitations of typical P4 switches. In particular, as the domain of packet ranks to be scheduled may be much greater than the number of registers available in a P4 implementation or even vary across configurations or time, it is not feasible to track the occurrences of packets on a per rank basis. Thus, an online algorithm based on the optimization problem introduced in 3.3 would necessarily have to be built on 3.3b. This version demands only n additional registers on top of the ones that store the queue bounds in order to learn the relative magnitudes of P_i for each queue over time.

3.3.1 Balancing the queues

As long as $n < k$ holds, any change to any of the queue bounds that reduces the expected inversion cost in a queue would result in an increase in one or more other queues. In other words, such an optimization step can only be performed as long as there are queues that can absorb the cost of taking in more ranks without them becoming a new, greater maximum cost queue. This means that our goal would be a state where the estimated costs of inversions in each of the queues are roughly the same.

Occasionally, a discrete problem can be made simpler by moving it to the domain of real numbers. Thus, for the sake of demonstration, let us pretend for a moment that ranks and queue bounds are actually real-valued. Let $f(x)$ denote the probability density function of incoming packet ranks and let $q_n = \infty$. When the optimization reaches a set of stable queue bounds, the following should hold:

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

Observe that moving q_i in the positive direction would decrease P_i and also increase P_{i-1}. Pushing in the opposite direction yields the opposite result. So, if we were in a suboptimal state before an optimization step, pushing q_i by $P_i - P_{i-1}$ should get us closer to the optimal one. This is somewhat reminiscent of a string of springs attached to each other, where each spring pulls at its neighbours with some force. The endpoints of the springs (like our queue bounds) move around according to the sum of the forces (like our values of P_i) acting upon them.

The number of packets received by each queue can be tracked in P4 trivially, and we can directly substitute such a counter for the values p_i. These counters, after all, hold the numerator for a fraction where the denominator is always the number of packets enqueued so far. As the denominator is shared across all of the counters, we can perform subtraction on them directly and treat the result as the force that acts upon the queue bound, as given in Fig. 3.5.

$$\Delta f(t) = \int_{q_i(t)}^{q_{i+1}(t)} f(x)dx - \int_{q_{i-1}(t)}^{q_i(t)} f(x)dx = p_i(t) - p_{i-1}(t)$$

$$q_i(t + \Delta t) = q_i(t) + \Delta f(t)$$

Figure 3.5: Per-packet optimization step for a spring-like heuristic. Here, Δt is a small constant time step.

3.3.2 Preliminary analysis on the convergence of the Spring heuristic

The optimization step as defined in Fig. 3.5 just happens to be identical to the Euler method for solving differential equations. So, if one were to plug in a cumulative density function $F(x)$ corresponding to some specific $f(x)$, it should be possible to look at the behaviour of the heuristic for that specific distribution as it converges the queue bounds to their final values if they do not match the measured input distribution:

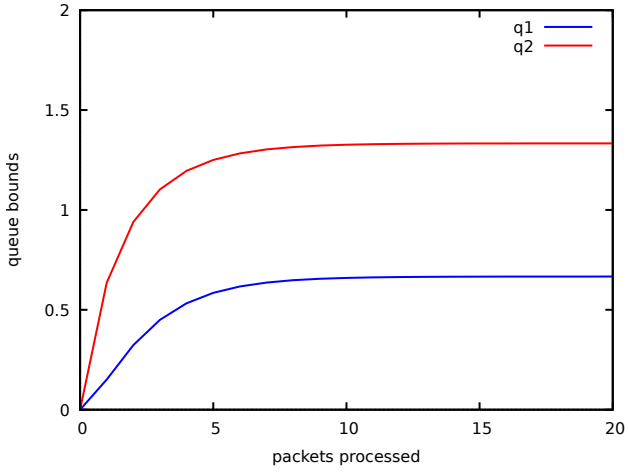
$$q_i' = \int_{q_i}^{q_{i+1}} f(x)dx - \int_{q_{i-1}}^{q_i} f(x)dx =$$
$$F(q_{i+1}) - F(q_i) - F(q_i) + F(q_{i-1}) =$$
$$F(q_{i+1}) - 2F(q_i) + F(q_{i-1})$$

Next, we evaluate the continuous model over some famous rank distributions. Since we intend to always have $q_0 = 0$, it is enough to evaluate the rest of the bounds of a system.

3.3.2.1 Uniform rank distribution over 3 queues

As mentioned earlier in Section 3.1.1, the optimal static queue bounds for a uniform packet distribution divide the range between 0 and k evenly. So on such an input, given a working implementation of the heuristic discussed so far, $n = 3$ queues, and $k = 2$, we should see the dynamic queue bounds converge around 2/3 and 4/3, respectively. This progression of queue bounds in time is depicted in Fig. 3.6.

$$f(x) = \max(1, x/k)$$
$$q_1' = F(q_2) - 2F(q_i)$$
$$q_2' = 1 - 2F(q_1) + F(q_1)$$



(a) Derivatives of the queue bounds over time.

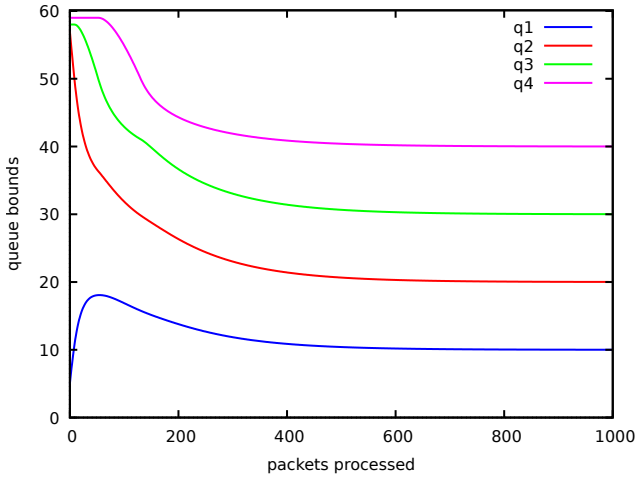
(b) Estimated behaviour of the heuristic over time, with initial queue bounds $q_1 = 0$, $q_2 = 0$

Figure 3.6: Model of the Spring heuristic on a uniform distribution of incoming packet ranks.

3.3.2.2 Uniform rank distribution over 5 queues

Let $n = 5$ and $k = 50$. In order to exercise the optimization, this time, some of the initial queue bounds start out above the maximum rank. As we can see in Fig. 3.7, q_0 temporarily rises above its own optimum to compensate for the high value of q_1, but eventually, all bounds settle on their optimums.

$$F(x) = \max(1, x/k)$$
$$q_1' = F(q_2) - 2F(q_i)$$
$$q_2' = F(q_3) - 2F(q_2) + F(q_1)$$
$$q_3' = F(q_4) - 2F(q_3) + F(q_2)$$
$$q_4' = 1 - 2F(q_4) + F(q_3)$$



(a) Derivatives of the queue bounds over time.

(b) Estimated behaviour of the heuristic over time, with initial queue bounds $q_1 = 5$, $q_2 = 57$, $q_3 = 58$ and $q_4 = 59$

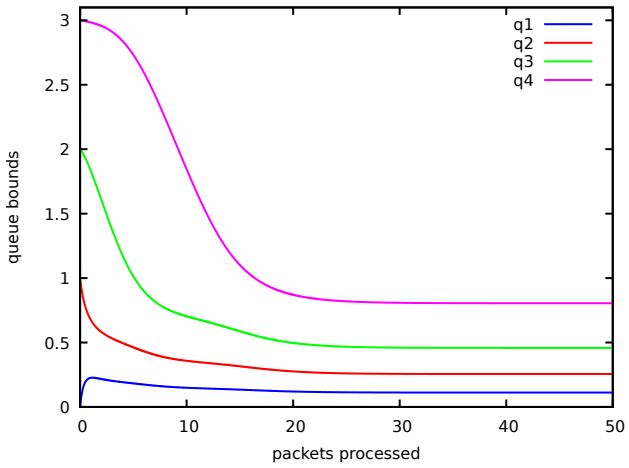
Figure 3.7: Model of the Spring heuristic on a uniform distribution of incoming packet ranks.

3.3.2.3 Exponential rank distribution

For exponential distributions, we should perceive the queue bounds converging densely on the lower range of the possible ranks and less so on the higher ones.

The optimal static bounds for an exponential input distribution using the cost metric of the Spring heuristic for each queue i would be ones where the cdf equals i/n, for $n = 5$ and $k = 2$ this roughly maps to $q_1 = 0.11$, $q_2 = 0.25$, $q_3 = 0.46$, and $q_4 = 0.8$. The trajectory of the system is depicted in Fig. 3.8.

$$F(x) = 1 - e^{-kx}$$
$$q_1' = F(q_2) - 2F(q_i)$$
$$q_2' = F(q_3) - 2F(q_2) + F(q_1)$$
$$q_3' = F(q_4) - 2F(q_3) + F(q_2)$$
$$q_4' = 1 - 2F(q_4) + F(q_3)$$



(a) Derivatives of the queue bounds over time.

(b) Estimated behaviour of the heuristic over time, with initial queue bounds $q_1 = 0$, $q_2 = 1$, $q_3 = 2$ and $q_4 = 3$

Figure 3.8: Model of the Spring heuristic on an exponential distribution of incoming packet ranks.

3.3.3 Forgetting past packets

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. This presents both the problem of potentially overflowing a finite size P4 register, as well as that of overlearning. If the distribution of ranks changes during the runtime of the algorithm, it would take a long time adapt because each new packet would only change the relative magnitudes of the packet counters by a minuscule amount.

Counting packets with *Exponentially Weighted Moving Averages* (EWMA) solves both problems at once. Let I_i be an indicator (taking on a value of either zero or one) of whether or not a newly received packet is assigned to the i-th queue. In addition, let us define a new parameter $0 < \alpha < 1$ 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 per-queue packet counters (μ) on each incoming packet as follows:

$$\mu_i \leftarrow (1 - \alpha)\mu_i + \alpha * I_i$$

It is easy to see that even if all packets were assigned to the same queue, the value of the associated packet counter would never exceed one.

3.3.4 On bound collisions

There is a slight catch with this approach: if we are not careful, the bound of a queue may end up being equal to or lower than that of its lower-ranked neighbour, as a result of the two bounds being close and being pushed in opposing directions.

Both cases seem unfortunate:

- If the bounds are equal, the SP-PIFO scheduling algorithm ends up using only the higher-ranked one, so we temporarily lose a queue, which results in more inversions.
- If the queue bounds trade places, lower-ranked packets end up in a higher-ranked queue and vice versa.

The latter would not only cause priority inversion between entire ranges of packet ranks, but it also invalidates an underlying assumption that the packet counters are related to queue-sized partitions in order. That would result in the packet counter of a new lower bound, higher-ranked queue pushing or pulling the bound of a now higher bound, lower-ranked queue from below (the opposite direction from the part of the distribution whence packets actually end up in that queue!).

To avoid these problems, we should implement an additional *collision* mechanism to the algorithm that ensures q_i is never pushed above $q_{i+1} - 1$, or below $q_{i-1} + 1$. These limits guarantee that queue bounds never swap places and that their rounded values are never equal so as to have all queues being able to receive packets in any intermediate configuration of the queue bounds.

3.3.5 Expected P4 compatibility

In terms of memory, the algorithm as described requires $3n$ registers, which should already be well within the capacity of most P4 compatible devices.

If SP-PIFO itself were adapted to use the same fixed-point number representation for the queue selection as used in the bounds adaptation phase, it should be possible to use as few as $2n$ registers, but this is an implementation detail that has no effect on the core idea itself. The simulations are all done using NetBench, where no strict memory limitations apply, and the SP-PIFO implementation there does not handle real-valued bounds, so the supplied pseudocode 1 does not make use of that observation.

Expressing the semantics might look like a potential issue, considering our use of real numbers and the fact that P4 only has integer types and limited support for operations on said integer types. As it turns out, not only are there already examples of fixed point arithmetics implemented in P4 [8], but there are also existing P4 implementations of EWMA itself [3]. Consequently, it should be possible to realize the algorithm in most P4 targets as is, implementing the main step with fixed-point arithmetics and setting the queue bounds to the rounded values the algorithm ends up with.

3.3.6 Static estimation of the Spring heuristic

While the idea behind the Spring heuristic was derived from the cost metrics defined in equations given in Fig. 3.2b and Fig. 3.3b, it does not really optimize for this exact same metric. In particular, it ignores the queue width and only accounts for the probability of incoming packets being mapped to each of the queues; see Fig. 3.9.

$$U_{\text{spring}}(Q_i) = P_i$$

$$U_{\text{spring}}(Q) = \max \sum_{i \in Q} U_{\text{spring}}(Q_i)$$

Figure 3.9: Underlying cost function of the Spring heuristic.

To have a baseline to compare the converged state of the Spring heuristic agaisnt when accompanied with EWMA, this cost function was also included in the aforementioned solver software that computes optimal static bounds.

3.3.7 Summary of the Spring heuristic

The pseudocode of the Spring heuristic is given in Alg. 1.

Algorithm 1: Spring heuristic: shift queue bounds with the goal of balancing the number of packets assigned to each queue.

```
begin
1   [q_0, ..., q_{n-1}] := [0, ..., n-1] // queue bounds as used by SP-PIFO
2   [r_0, ..., r_{n-1}] := [0, ..., n-1] // queue bounds, represented as
        fixed-point numbers
3   [μ_0, ..., μ_{n-1}] := [0, ..., 0] // metric: amort. # packets enqueued in
        Q[i] or amort. metric of inversions in Q[i]
    while Packet arrives with rank j do
4       Packet j is enqueued into queue Q[i] s.t. j ≥ q_i, where i is the greatest
          queue index satisfying this condition.
        // Refreshing metrics
5       [μ_0, ..., μ_{n-1}] := [μ_0, ..., μ_{n-1}] * (1 − α)
6       μ_i := μ_i + α
        // Updating bounds
        for i = n − 1, ..., 1 do
7           upperBound := +∞
8           lowerBound := r_{i−1} + 1
9           if i < n − 1 then
                upperBound := r_{i+1} − 1
10          r_i := r_i + μ_i − μ_{i−1}
11          r_i := min {max {lowerBound, r_i}, upperBound}
12          q_i := round(r_i)
```

Chapter 4

Implementation

4.1 About NetBench

NetBench [7] is an event-driven packet simulation framework, written in Java. It allows researchers to perform experiments involving custom protocols or scheduling algorithms in a reproducible manner, and it was used as a benchmarking tool for both the inversion count and flow completion time measurements of [1]. As there already was a version of NetBench that contained a reference implementation of SP-PIFO it made sense to reuse it, to make the measurements reproducible and comparable to earlier work as much as possible.

4.2 NetBench modifications

To facilitate the measurements of this study, some changes have been performed on the codebase.

4.2.1 Separation of the implementation of PUPD from SP-PIFO

The upstream version of NetBench did not conceptually separate adaptation functions from the core SP-PIFO scheduling behaviour, and as such, the code did not lend itself readily for plugging in a different adaptation algorithm.

This shortcoming was tackled by extracting the PUPD implementation into a different Java class and replacing all its previously inlined functionality with calls to a newly introduced `AdaptationAlgorithm` interface, with a concrete implementation now injected into SP-PIFO queues at construction time.

Unit tests have been written against the original SP-PIFO behaviour, then subsequently ported to the new implementation to reduce the risk of unintended changes to the behaviour of an SP-PIFO/PUPD system.

4.2.2 Implementation of new adaptation algorithms

With SP-PIFO being capable of using any caller-supplied adaptation algorithm, I could now write a clean implementation for the algorithms proposed in the previous sections. In particular, I implemented the optimal static algorithm outside NetBench, and I wrote an

adaptation class in NetBench that can take static queue bounds from an external input, and I implemented a version of the Spring heuristic entirely in NetBench.

4.3 NetBench parallelization

As of the latest commit at the time of the study, the upstream version of NetBench only had the stem of a command line interface to control its various runtime parameters, but the various pieces had not been wired up, and the final product was not functional; perhaps due to this reason, the measurements in [1] were all driven entirely from Java code in a completely serial manner. This left the computational resources provided by multi-core CPUs unused.

By filling out the missing details, it became possible to drive NetBench simulations and tune parameters from external scripts. This opens up a number of new approaches, from easy automatic test case generation in any language to driving simulations with tools capable of executing them in parallel, such as GNU `parallel`. Porting the original measurements of [1] to shell scripts invoking `parallel` resulted in a near-linear speedup with regards to the number of CPU cores present on the test system.

Chapter 5

Evaluation

In this chapter, the goal is to evaluate the utility of the Spring heuristic. The following questions, in particular, are essential to this goal:

- Does the Spring heuristic converge even when equipped with EWMA?
- If so, how many steps does it take to reach a stable state?
- Does the Spring heuristic converge around the expected static bounds, as determined by the optimal algorithm?
- How well does it perform with regards to the number of inversions on its output?
- How well does it perform with regards to the total cost of inversions on its output?

Since the Spring heuristic aims to approximate a set of static bounds, the questions that relate to inversions also apply to the static bounds.

5.1 Simulation parameters

The simulations used the upstream traffic generators from NetBench, labeled **exponential**, and **inverse_exponential** respectively. These components generate random numbers from an exponential distribution ($\lambda = \frac{1}{25}$), map them to integer values in $[0, 100)$, and assign them to the generated packets as ranks. The only major difference between these components is that **inverse_exponential** also subtracts the generated integer from the maximum rank, yielding a distribution that is very similar to the original but places the common ranks at the top of their range instead of the bottom.

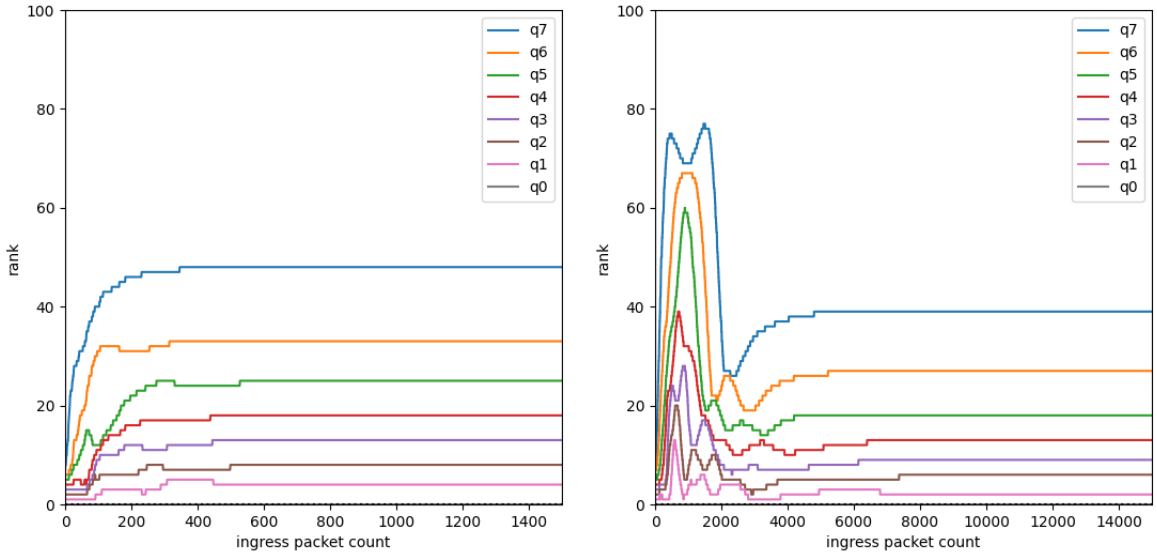
The configurations of the PUPD and Greedy Gradient algorithms match those used in the inversion-related measurements of [1]: $n = 8$ and $k = 100$. The Spring Heuristic uses the same parameters as PUPD, with the addition of an α parameter to tune the EWMA component.

All NetBench measurements were configured with a one second limit on the simulated runtime, which results in around one million packets.

5.2 Evaluation results

5.2.1 Stability

First, we present a series of simulations to understand whether the Spring heuristic reaches a static steady state in limited time. Ideally, stability should be attained for any set of initial queue bounds; below we start the algorithm from the queue bounds 1, 2, ..., k but we note that similar results were obtained for any initial queue bound setting, as long as the each queue bounds was greater than or equal to the preceding one.



(a) $\alpha = 0.1$ **(b)** $\alpha = 0.01$

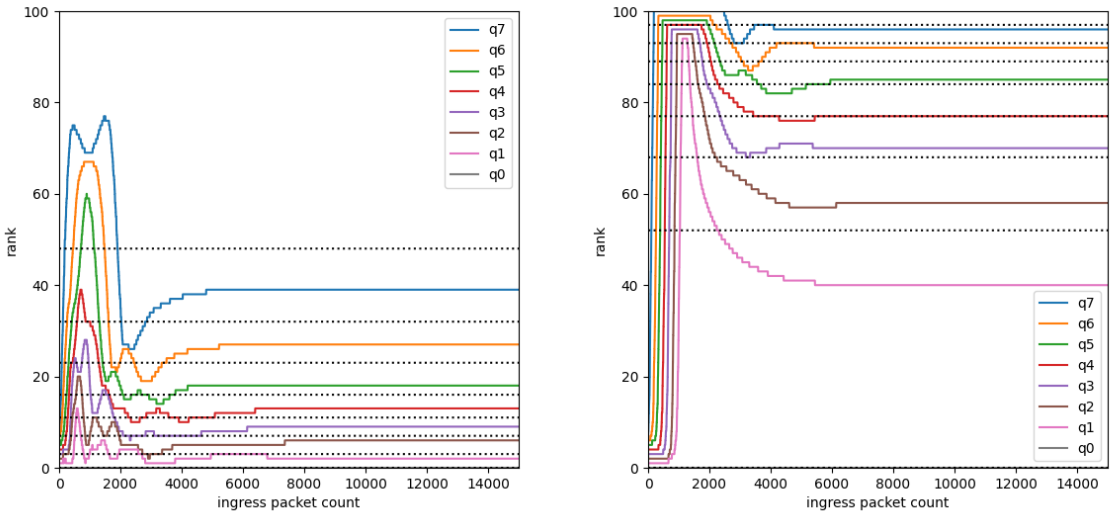
Figure 5.1: Convergence of the Spring heuristic on **exponential** packet ranks for different values of α.

As it can be seen in Figure 5.1a and 5.1b, the Spring heuristic still converges on different input rank distributions, even when EWMA is used in place of a CDF.

The rate of convergence seems to depend on the value of the α parameter, with smaller values consistently resulting in a longer learning phase (note the different scale on the X axes in Figure 5.1a and Figure 5.1b). Empirically $\alpha = 0.01$ seems to yield reasonable results, and for this reason was chosen as the parameter to evaluate the Spring heuristic in the following sections.

5.2.2 Convergence to the optimal bounds

While we already showed how the optimization step converges to the estimated optimal queue bounds earlier in section 3.3.2, we had at that time assumed a perfect knowledge of the input rank CDF $F(x)$, which is replaced by a per-queue EWMA in the current version of the algorithm. As such, we should check whether the algorithm still converges to the expected bounds.



(a) exponential rank distribution.

(b) inverse_exponential rank distribution.

Figure 5.2: Convergence of the Spring heuristic compared against the optimal static bounds of U_{spring}, as defined in equation 3.9. The dotted lines depict the static solution to the cost function, the spring heuristic was configured with $\alpha = 0.01$

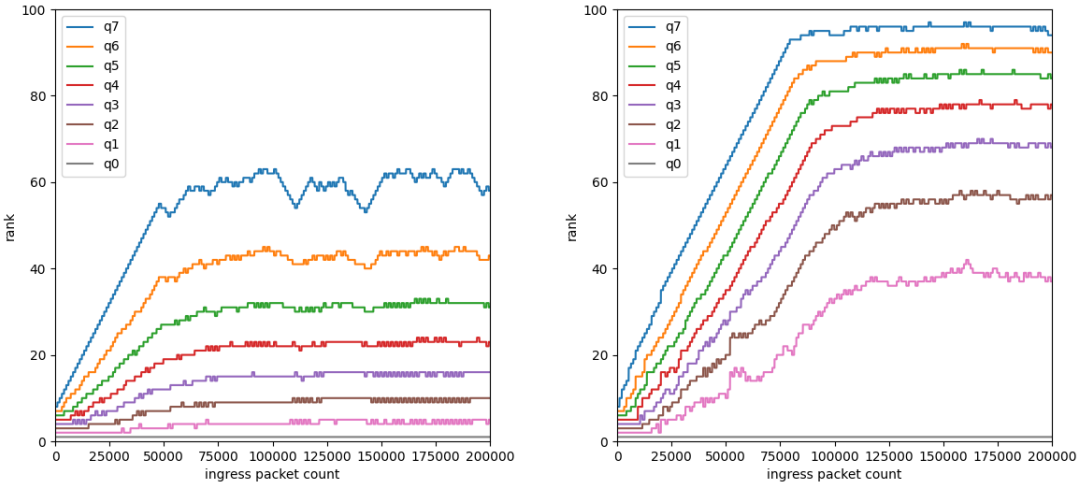
Figure 5.2a and Figure 5.2b show the optimal static bounds for the same cost function of the Spring heuristic, as well as the as the bounds assigned by the NetBench implementation of the Spring heuristic as packets are processed on its input.

As we can see, the stable state of the Spring heuristic can result in either optimal or close to optimal bounds for frequently used queues. The difference between the optimal static bounds and the values converged to by the Spring heuristic might be a result of the use of EWMA, but optimizations to reduce the difference have not been evaluated yet as a part of this study.

5.2.3 Rate of convergence

As there are not many other dynamic, converging adaptation algorithms implemented for SP-PIFO yet, the only candidate to compare the rate of convergence to is the Greedy Gradient heuristic.

As it can be seen in Figure 5.2a and Figure 5.2b, the queue bounds, apart from an initial learning phase, are completely stable over the rest of the duration of the simulation. This heuristic may converge around a stable state in as few as 8000 packets.



(a) Queue bounds over time for the Greedy heuristic, with an **exponential** input rank distribution.

(b) Queue bounds over time for the Greedy heuristic, with an **inverse_exponential** input rank distribution.

Figure 5.3: Convergence of the Greedy Gradient heuristic.

Figure 5.3a and Figure 5.3b show the behaviour of the Greedy Gradient heuristic on the same input distributions. While the queue bounds do converge around a set of values, they take more optimizations steps (performed on each incoming packet) than their counterparts controlled by the Spring heuristic.

5.3 Evaluation of inversions

Judging by the relative speed of convergence, the Spring heuristic may be well suited to quickly changing network traffic. Next, we conduct a controlled set of experiments in order to understand how well the Spring heuristic approximates PIFO-like behaviour, comparing it against the alternatives based on the number and intensity of inversions it introduces. The particular algorithms we compare as follows:

- **Spring**: This algorithm is the Spring heuristic, configured with $\alpha = 0.01$.
- **Static**: This algorithm utilizes static bounds, configured to the optimal bounds of the cost function of the Spring heuristic, as defined in equation 3.9. This algorithm serves as a baseline for the **Spring** heuristic, as it represents its optimal state (zero time required for convergence and perfect knowledge of the cumulative density function).
- **Greedy**: The original configuration used for the analysis of inversions in the Greedy Gradient heuristic in [1] has been reused for this measurement. Here, it serves as a reference, being the only other eventually convergent dynamic adaptation algorithm that is currently implemented for SP-PIFO.
- **PUPD**: The original Push-up-Push-Down algorithm as proposed in [1]. It is noteworthy for being much simpler than the Greedy Gradient heuristic and thus easier to implement in a programmable switch. It is currently the only SP-PIFO adaptation algorithm to have been implemented in P4.

Table 5.1 specifies the total number of inversions for each evaluated SP-PIFO adaptation algorithm.

Table 5.1: Total number of inversions, and total cost of inversions over a second of simulated traffic for each of the simulated input rank distributions, when SP-PIFO is configured to use a specific adaptation algorithm.

Algorithm	**exponential** distribution		**inverse_exponential** distribution	
	Inversion count	$\sum$ Inversion cost	Inversion count	$\sum$ Inversion cost
Static	194739	1169624	172265	641884
Greedy	197956	1266415	235268	1678639
Spring	209236	2000814	203394	602474
PUPD	248957	3029313	367048	5215126

We observe that the Spring heuristic seems to perform similarly to the Greedy Gradient heuristic in terms of total inversion count. For total inversion cost it perform slightly worse than the Greedy Gradient heuristic over the **exponential** distribution, but it still avoids around 30% of the inversion cost of PUPD. At the same time, it performs remarkably well in the case of the **inverse_exponential** rank distribution. This behaviour is likely related to the fact that in this distribution, the most common ranks are those at the top of their range rather than at the bottom. When earlier the topmost packet counters did not seem to retain a high enough value to raise the bounds, it resulted in the associated queues receiving more packets as a result.

A similar phenomenon occurs in the lower-ranked queues in the case of the **inverse_exponential** distribution, but this time, when q_i fails to pull q_{i-1} high enough, q_{i-1} is left in a position where it divides the remaining ranks more evenly, limiting the maximum cost of inversion that may happen in that queue. Since the affected queues are also the ones with the least amount of traffic, the difference does not seem to noticeably affect the number of inversions by the end of the simulation.

The per-rank distribution of inversions is given next. In particular, Figure 5.4 specifies the per-rank inversion counts and Figure 5.5 shows the per-rank total cost of inversions for the **exponential** distribution. These results seem to support the earlier argumentation: the highest ranked queue appears to cause noticeably more inversions with the Spring heuristic than it does with other adaptation algorithms, and the total cost of inversions also increases for these ranks.

In the case of the **inverse_exponential** distribution, while there is still a slight increase in the number of inversions in the lower-ranked queues (see Figure 5.6), it is noticeably lower than what could be observed in the highest-ranked queue during the previous simulation, and its effect on the total cost of inversions (Figure 5.7) is also much less severe.

This, and the results showing the **Static** algorithm performing well in both metrics with both input distributions, further suggests that improving on packet counting mechanism, either by slightly modifying or replacing EWMA may improve the behaviour of the Spring heuristic in the general case.

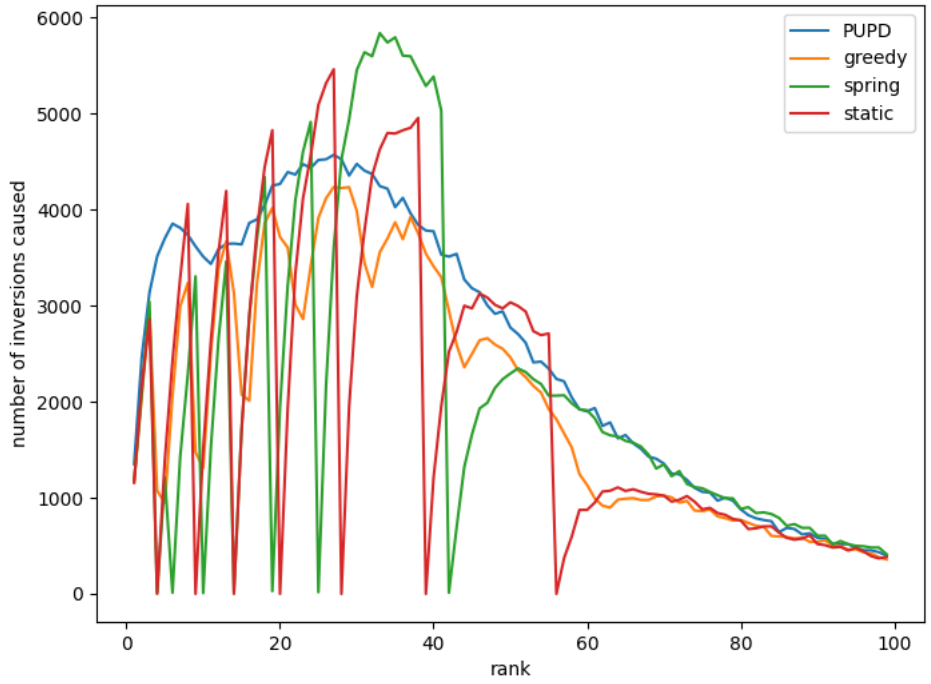


Figure 5.4: Total number of inversions, per-rank, for an **exponential** input rank distribution over the duration of one second.

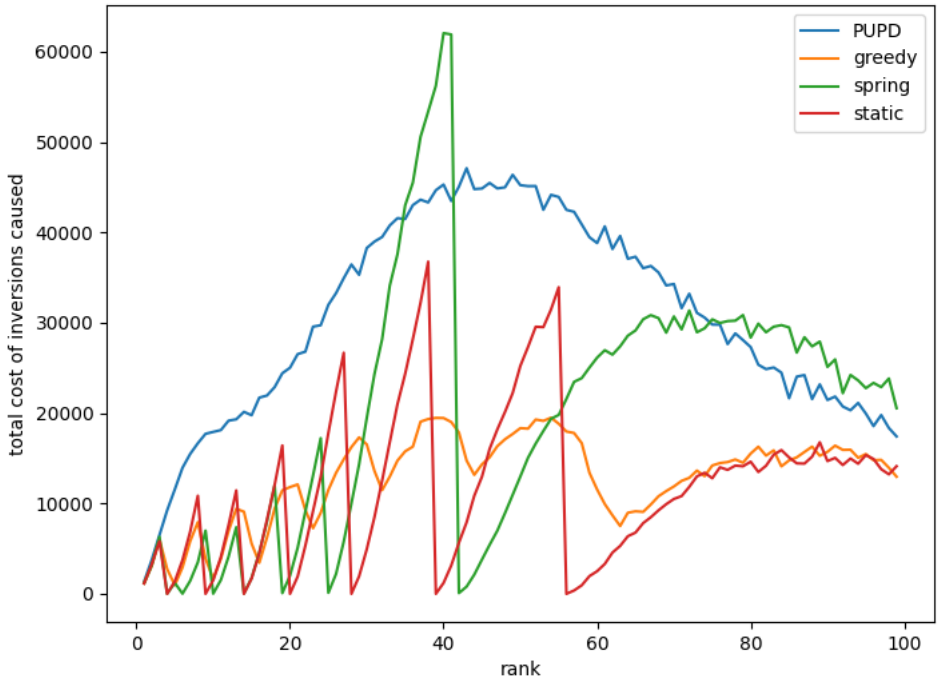


Figure 5.5: Total cost of inversions, per-rank, for an **exponential** input rank distribution over the duration of one second.

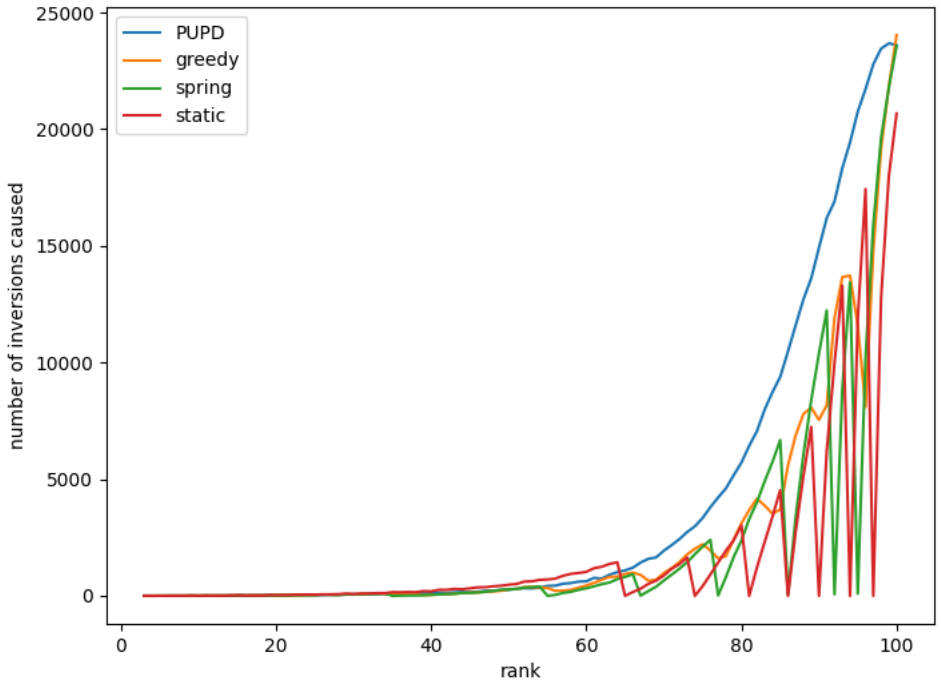


Figure 5.6: Total number of inversions, per-rank, for an **inverse_exponential** input rank distribution over the duration of one second.

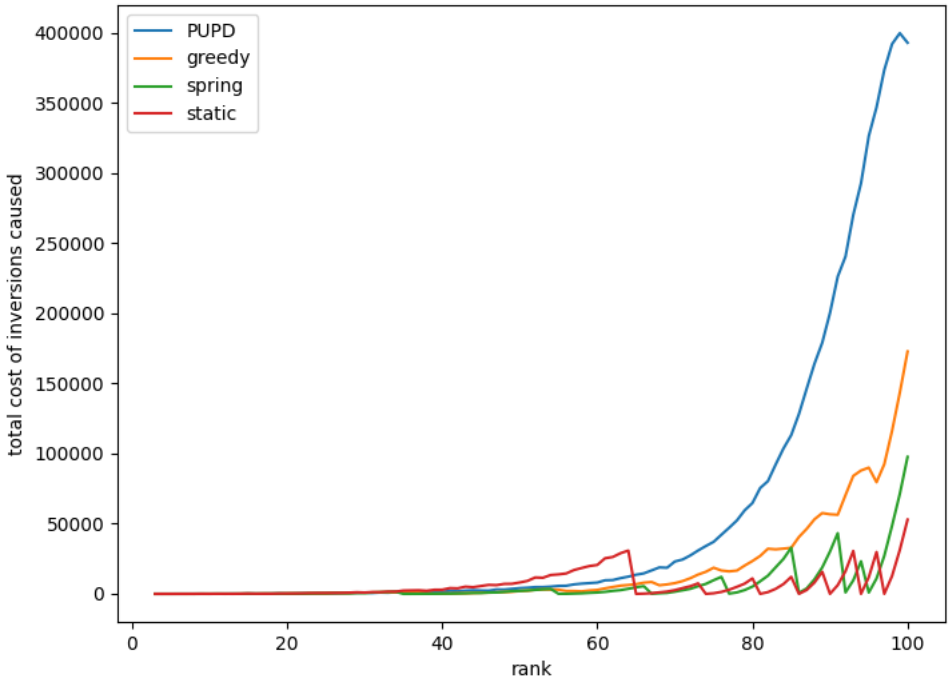


Figure 5.7: Total cost of inversions, per-rank, for an **inverse_exponential** input rank distribution over the duration of one second.

Chapter 6

Conclusion

Motivated by the strong industry trends towards making the entire network data plane programmable, in this study, we evaluated various approaches to make the final missing piece of the general packet processing pipeline, namely packet schedulers, reconfigurable. We built on existing proposals from recent literature: PIFO provides the basic abstractions for a building programmable queuing algorithm but it is challenging to implement in hardware, whereas SP-PIFO provides a viable path towards emulating PIFO with **existing** commercial off-the-shelf switches using a clever adaptation algorithm that arbitrates packets between a small set of priority queues.

Our main goal was to understand the optimality of the SP-PIFO adaptation mechanisms described in the literature. For this, we proposed the first optimal algorithm to compute the best static bounds for a given inversion cost function and rank distribution. The algorithm runs in polynomial time: its complexity is $O(kn^2)$ where k is the maximum rank and n is the number of queues. The algorithm was implemented in a standalone program, and the output of this program was used as a baseline during NetBench simulations. We observed that, the optimal static bounds provide the best results, i.e., the lowest inversion count, among the evaluated SP-PIFO adaptation algorithms.

Unfortunately, it is impossible to implement the optimal static bounds algorithm in P4. Inspired by springs and an inversion cost metric that does not require tracking a large number of variables, a new online bounds adaptation heuristic was designed. The proposed heuristic, depending on its configuration, may be able to adapt to changes in network traffic quickly, while it also yields stable queue bounds. Crucially, the new heuristic is easy to reason about formally, which is not the case for, e.g., PUPD, and it provides somes favourable results when compared to the existing solutions. Alas, it does not perform as well as it theoretically could in terms of total inversion cost, but it shows great promise; our study also provides some insight into its weaknesses that may aid in further improving it.

While the proposed Spring heuristic can already perform well, there are a number of remaining issues and open future questions:

- How to choose an optimal value for α?
- How quickly does it adapt when the rank distribution changes after the algorithm already settled on a set of queue bounds?
- Is it possible to compensate for the distorted view of the rank distribution in the least frequently utilized queues while still relying on EWMA?

- If it is not, are there any alternatives to EWMA that can be implemented in P4?
- While the Spring heuristic is currently believed to P4 compatible, no P4 implementation has been created yet.
- This study only focused on inversions, but SP-PIFO is merely a tool, not a goal. Ultimately, users would like to make use of PIFO-like building blocks for their own algorithms. The Spring heuristic has yet to be evaluated as an element of such a larger system.

Bibliography

[1] Albert Gran Alcoz, Alexander Dietmüller, and Laurent 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)*, pages 59–76, 2020. URL https://www.usenix.org/conference/nsdi20/presentation/alcoz.

[2] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, August 2013. ISSN 0146-4833. DOI: 10.1145/2534169.2486031. URL https://doi.org/10.1145/2534169.2486031.

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

[4] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1): 87–90, 1958.

[5] Kevin Dooley and Ian J. Brown. *Cisco cookbook*. 2003. URL http://www.oreilly.com/catalog/9780596003678.

[6] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411. DOI: 10.1145/321738.321743. URL https://doi.org/10.1145/321738.321743.

[7] G. Memik, W.H. 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)*, pages 39–42, 2001. DOI: 10.1109/ICCAD.2001.968595.

[8] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 85–98, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346535. DOI: 10.1145/3098822.3098829. URL https://doi.org/10.1145/3098822.3098829.

[9] Youhwan Seol, Doyeon Hyeon, Junhong Min, Moonbeom Kim, and Jeongyeup Paek. Timely survey of time-sensitive networking: Past and future directions. *IEEE Access*, 9:142506–142527, 2021. DOI: 10.1109/ACCESS.2021.3120769.

[10] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 44–57, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341936. DOI: 10.1145/2934872.2934899. URL https://doi.org/10.1145/2934872.2934899.

[11] George Varghese. Scheduling packets. In George Varghese, editor, *Network Algorithmics*, The Morgan Kaufmann Series in Networking, pages 339–361. Morgan Kaufmann, San Francisco, 2005. DOI: https://doi.org/10.1016/B978-012088477-3/50017-5. URL https://www.sciencedirect.com/science/article/pii/B9780120884773500175.