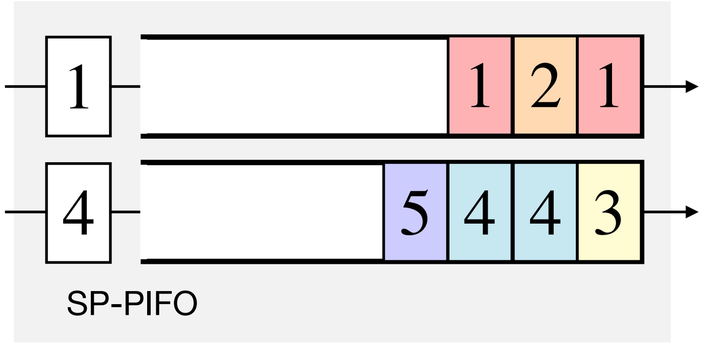
SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues



Albert Gran Alcoz,
Alexander Dietmüller,
Laurent Vanbever

sp-pifo.ethz.ch

NSDI '20
February, 25 2020

ETH zürich


144179.144199.pdf (page 1 of 13)

Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism

David D. Clark[1]
Laboratory for Computer Science
Massachusetts Institute of Technology
ddc@lcs.mit.edu

Scott Shenker Lixia Zhang
Palo Alto Research Center
Xerox Corporation
shenker, lixia@parc.xerox.com

Abstract

This paper considers the support of real-time applications in an *Integrated Services Packet Network* (ISPN). We first review the characteristics of real-time applications. We observe that, contrary to the popular view that real-time applications necessarily require a fixed delay bound, some real-time applications are more flexible and can adapt to current network conditions. We then propose an ISPN architecture that supports two distinct kinds of real-time service: *guaranteed* service, which is the traditional form of real-time service discussed in most of the literature and involves pre-computed worst-case delay bounds, and *predicted* service which uses the measured performance of the network in computing delay bounds. We then propose a packet scheduling mechanism that can support both of these real-time services as well as accommodate datagram traffic. We also discuss two other aspects of an overall ISPN architecture: the service interface and the admission control criteria.

1 Introduction

The current generation of telephone networks and the current generation of computer networks were each designed to carry specific and very different kinds of traffic: analog voice and digital data. However, with the digitizing of telephony in ISDN and the increasing use of multi-media in computer applications, this distinction is rapidly disappearing. Merging these sorts of services into a single network, which we refer to here as an *Integrated Services Packet Network* (ISPN), would yield a single telecommunications infrastructure offering a multitude of advantages, including vast economies of scale, ubiquity of access, and improved statistical multiplexing. There is a broad consensus, at least in the computer networking community, that an ISPN is both a worthy and an achievable goal. However, there are many political, administrative, and technical hurdles to overcome before this vision can become a reality.

One of the most vexing technical problems that blocks the path towards an ISPN is that of supporting real-time applications in a packet network. Real-time applications are quite different from standard data applications, and require service that cannot be delivered within the typical data service architecture. In Section 2 we discuss the nature of real-time applications at length; here, however, it suffices to observe that one salient characteristic of the real-time applications we consider is that they require a bound on the delivery delay of each packet[2]. While this bound may be statistical, in the sense that some small fraction of the packets may fail to arrive by this bound, the bound itself must be known *a priori*. The traditional data service architecture underlying computer networks has no facilities for prescheduling resources or denying service upon overload, and thus is unable to meet this real-time requirement.

Therefore, in order to handle real-time traffic, an enhanced architecture is needed for an ISPN. We identify four key components to this architecture. The first piece of the architecture is the nature of the commitments made by the network when it promises to deliver a certain quality of service. We identify two sorts of commitments, *guaranteed* and *predicted*. Predicted service is a major aspect of our paper. While the idea of predicted service has been considered before, the issues that surround it have not, to our knowledge, been carefully explored.

The second piece of the architecture is the service interface, i.e., the set of parameters passed between the source and the network. The service interface must include both the characterization of the quality of service the network will deliver, fulfilling the need of applications to know when their packets will arrive, and the characterization of the source's traffic, thereby allowing the network to knowledgeably allocate resources. In this paper we attempt to identify the critical aspects of the service interface, and offer a particular interface as an example. We address in passing the need for enforcement of these characterizations.

The third piece of the architecture is the packet scheduling behavior of network switches needed to meet these service commitments. We discuss both the actual scheduling algorithms to be used in the switches, as well as the scheduling information that must be carried in packet headers. This

[1]Research at MIT was supported by DARPA through NASA Grant NAG 2-582, by NSF grant NCR-8814187, and by DARPA and NSF through Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1992 ACM 0-89791-526-7/92/0008/0014...$1.50

[2]Since the term *bound* is tossed around with great abandon in the rest of the paper, we need to identify several different meanings to the term. As a *priori* bound on delay is a statement that none of the future delays will exceed that amount. A *post facto* bound is the maximal value of a set of observed delays. Statistical bounds allow for a certain percentage of violations of the bound; absolute bounds allow none.

14


nsdi15-paper-bai.pdf (page 2 of 15)

Information-Agnostic Flow Scheduling for Commodity Data Centers

sigcomm13-pfabric.pdf (page 1 of 12)

pFabric: Minimal Near-Optimal Datacenter Transport

Mohammad Alizadeh[†‡], Shuang Yang[†], Milad Sharif[†], Sachin Katti[†],
Nick McKeown[†], Balaji Prabhakar[*], and Scott Shenker[§]

[†]Stanford University [‡]Insieme Networks [*]U.C. Berkeley / ICSI
{alizade, shyang, msharif, skatti, nickm, balaji}@stanford.edu shenker@icsi.berkeley.edu

ABSTRACT

In this paper we present pFabric, a minimalistic datacenter transport design that provides near theoretically optimal flow completion times even at the 99th percentile for short flows, while still minimizing average flow completion time for long flows. Moreover, pFabric delivers this performance with a very simple design that is based on a key conceptual insight: datacenter transport should decouple flow scheduling from rate control. For flow scheduling, packets carry a single priority number set independently by each flow; switches have very small buffers and implement a very simple priority-based scheduling/dropping mechanism. Rate control is also correspondingly simpler; flows start at line rate and throttle back only under high and persistent packet loss. We provide theoretical intuition and show via extensive simulations that the combination of these two simple mechanisms is sufficient to provide near-optimal performance.

Categories and Subject Descriptors: C.2.1 [Computer-Communication Networks]: Network Architecture and Design
General Terms: Design, Performance
Keywords: Datacenter network, Packet transport, Flow scheduling

1. INTRODUCTION

Datacenter workloads impose unique and stringent requirements on the transport fabric. Interactive soft real-time workloads such as the ones seen in search, social networking, and retail generate a large number of small requests and responses across the datacenter that are stitched together to perform a user-requested computation (e.g., delivering search results). These applications demand low latency for each of the short request/response flows, since user-perceived performance is dictated by how quickly responses to all (or a large fraction of) the requests are collected and delivered back to the user. However in currently deployed TCP-based fabrics, the latency for these short flows is poor — flow completion times (FCT) can be as high as tens of milliseconds while in theory these flows could complete in 10-20 microseconds. The reason is that these flows often get queued up behind bursts of packets from large flows of co-existing workloads (such as backup, replication, data mining, etc) which significantly increases their completion times.

Motivated by this observation, recent research has proposed new datacenter transport designs that, broadly speaking, use rate control to reduce FCT for short flows. One line of work [3, 4] improves FCT by keeping queues near empty through a variety of mechanisms (adaptive congestion control, ECN-based feedback, pacing, etc) so that latency-sensitive flows see small buffers and consequently small latencies. These *implicit* techniques generally improve FCT for short flows but they can never precisely determine the right flow rates to optimally schedule flows. A second line of work [21, 14] *explicitly* computes and assigns rates from the network to each flow in order to schedule the flows based on their sizes or deadlines. This approach can potentially provide very good performance, but it is rather complex and challenging to implement in practice because accurately computing rates requires detailed flow state at switches and also coordination among switches to identify the bottleneck for each flow and avoid under-utilization (§2).

Our goal in this paper is to design the simplest possible datacenter transport scheme that provides near-optimal flow completion times, even at the 99th percentile for latency-sensitive short flows. To this end, we present pFabric,[1] a minimalistic datacenter fabric whose entire design consists of the following:

- End-hosts put a single number in the header of every packet that encodes its priority (e.g., the flow's remaining size, deadline). The priority is set independently by each flow and no coordination is required across flows or hosts to compute it.
- Switches are simple; they have very small buffers (e.g., 36KB per port in our evaluation) and decide which packets to accept into the buffer and which ones to schedule strictly according to the packet's priority number. When a new packet arrives and the buffer is full, if the incoming packet has lower priority than all buffered packets, it is dropped. Else, the lowest priority packet in the buffer is dropped and replaced with the incoming packet. When transmitting, the switch sends the packet with the highest priority. Thus each switch operates independently in a greedy and local fashion.
- Rate control is minimal; all flows start at line-rate and throttle their sending rate only if they see high and persistent loss. Thus rate control is lazy and easy to implement.

pFabric thus requires no flow state or complex rate calculations at the switches, no large switch buffers, no explicit network feedback, and no sophisticated congestion control mechanisms at the end-host. pFabric is a clean-slate design; it requires modifications both at the switches and the end-hosts. We also present a preliminary design for deploying pFabric using existing switches, but a full design for incremental deployment is beyond the scope of this paper.

[1]pFabric was first introduced in an earlier paper [5] which sketched a preliminary design and initial simulation results.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM'13, August 12–16, 2013, Hong Kong, China.
Copyright 2013 ACM 978-1-4503-2056-6/13/08 ...$15.00.


00234856.pdf (page 1 of 14)

A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case

nsdi18-sharma.pdf (page 2 of 17)

Approximating Fair Queueing on Reconfigurable Switches

Naveen Kr. Sharma[*] *Ming Liu*[*] *Kishore Atreya*[†] *Arvind Krishnamurthy*[*]

[*]University of Washington
[†]Cavium Inc.

Abstract

Congestion control today is predominantly achieved via end-to-end mechanisms with little support from the network. As a result, end-hosts must cooperate to achieve optimal throughput and fairness, leading to inefficiencies and poor performance isolation. While router mechanisms such as Fair Queuing guarantee fair bandwidth allocation to all participants and have proven to be optimal in some respects, they require complex flow classification, buffer allocation, and scheduling on a per-packet basis. These factors make them expensive to implement in high-speed switches.

In this paper, we use emerging reconfigurable switches to develop an approximate form of Fair Queueing that operates at line-rate. We leverage configurable per-packet processing and the ability to maintain mutable state inside switches to achieve fair bandwidth allocation across all traversing flows. Further, present our design for a new dequeuing scheduler, called Rotating Strict Priority scheduler that lets us transmit packets from multiple queues in approximate sorted order. Our hardware emulation and software simulations on a large leaf-spine topology show that our scheme closely approximates ideal Fair Queueing, improving the average flow completion times for short flows by 2-4x and 99th tail latency by 4-8x relative to TCP and DCTCP.

1 Introduction

Most current congestion control schemes rely on end-to-end mechanisms with little support from the network (e.g., ECN, RED). While this approach simplifies switches and lets them operate at very high speeds, it requires end-hosts to cooperate to achieve fair network sharing, thereby leading to inefficiencies and poor performance isolation. On the other hand, if the switches were capable of maintaining per-flow state, extracting rich telemetry from the network, and performing configurable per-packet processing, one can realize intelligent congestion control mechanisms that take advantage of dynamic network state directly inside the network and improve network performance.

One such mechanism is Fair Queueing, which has been studied extensively and shown to be optimal in several aspects. It provides the illusion that every flow (or participant) has its own queue and receives a fair share of the bandwidth under all circumstances, regardless of other network traffic. Having the network enforce fair bandwidth allocation offers several benefits. It simplifies congestion control at the end-hosts, removing the need to perform slow-start or complex congestion avoidance strategies. Further, flows can ramp up quickly without affecting other network traffic. It also provides strong isolation among competing flows, protects well-behaved flows from ill-behaving traffic, and enables bounded delay guarantees [34].

A fair bandwidth allocation scheme is potentially well suited to today's datacenter environment, where multiple applications with diverse network demands often co-exist. Some applications require low latency, while others need sustained throughput. Datacenter networks must also contend with challenging traffic patterns – such as large incasts or fan-in, micro-bursts, and synchronized flows, – which can all be managed effectively using a fair queueing mechanism. Fair queueing mechanisms can also provide bandwidth guarantees for multiple tenants of a shared cloud infrastructure [35].

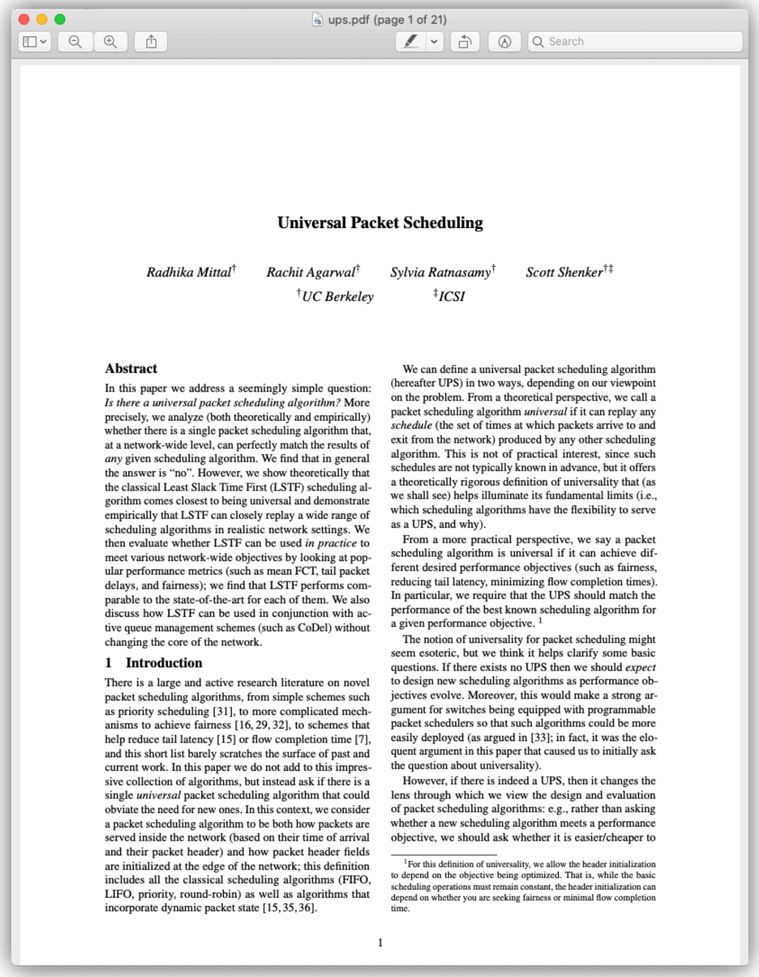
Over the years, several algorithms for enforcing fair bandwidth allocation have been proposed [25, 27, 28, 33], but rarely deployed in practice, primarily due to their inherent complexities. These algorithms maintain state and perform operations on a per-flow basis, making them challenging to implement at data rates of 3-6 Tbps in hardware. However, recent advances in switching hardware allow flexible per-packet processing and the ability to maintain limited mutable state at switches without sacrificing performance [12, 6]. In this paper, we explore whether an efficient fair queueing implementation can be realized using these emerging reconfigurable switches.

We present *Approximate Fair Queueing (AFQ)*, a fair bandwidth allocation mechanism that approximates the various components of an ideal fair queueing scheme using features available in emerging programmable switches, such as the ability to maintain and mutate switch state on a per-packet basis, perform limited computation for each packet, and dynamically determine which egress queue to use for a given packet. We describe a variant of the packet-pair flow control protocol [24], designed to work with AFQ, that achieves close to optimal performance while maintaining short queues. We further prototype an AFQ implementation on a Cavium networking processor and study its feasibility on upcoming reconfigurable switches. Using a real hardware testbed and large-scale simulations, we demon-

Minimize tail packet delays
[SIGCOMM '92]

Minimize flow completion times
[SIGCOMM '13, NSDI '15]

Enforce max-min fairness
[ToN '93, NSDI '18]



ups.pdf (page 1 of 21)

Search

Universal Packet Scheduling

Radhika Mittal[†] *Rachit Agarwal*[†] *Sylvia Ratnasamy*[†] *Scott Shenker*[‡‡]
[†]*UC Berkeley* [‡]*ICSI*

Abstract

In this paper we address a seemingly simple question: *Is there a universal packet scheduling algorithm?* More precisely, we analyze (both theoretically and empirically) whether there is a single packet scheduling algorithm that, at a network-wide level, can perfectly match the results of *any* given scheduling algorithm. We find that in general the answer is "no". However, we show theoretically that the classical Least Slack Time First (LSTF) scheduling algorithm comes closest to being universal and demonstrate empirically that LSTF can closely replay a wide range of scheduling algorithms in realistic network settings. We then evaluate whether LSTF can be used *in practice* to meet various network-wide objectives by looking at popular performance metrics (such as mean FCT, tail packet delays, and fairness); we find that LSTF performs comparable to the state-of-the-art for each of them. We also discuss how LSTF can be used in conjunction with active queue management schemes (such as CoDel) without changing the core of the network.

1 Introduction

There is a large and active research literature on novel packet scheduling algorithms, from simple schemes such as priority scheduling [31], to more complicated mechanisms to achieve fairness [16, 29, 32], to schemes that help reduce tail latency [15] or flow completion time [7], and this short list barely scratches the surface of past and current work. In this paper we do not add to this impressive collection of algorithms, but instead ask if there is a single *universal* packet scheduling algorithm that could obviate the need for new ones. In this context, we consider a packet scheduling algorithm to be both how packets are served inside the network (based on their time of arrival and their packet header) and how packet header fields are initialized at the edge of the network; this definition includes all the classical scheduling algorithms (FIFO, LIFO, priority, round-robin) as well as algorithms that incorporate dynamic packet state [15, 35, 36].

We can define a universal packet scheduling algorithm (hereafter UPS) in two ways, depending on our viewpoint on the problem. From a theoretical perspective, we call a packet scheduling algorithm *universal* if it can replay any *schedule* (the set of times at which packets arrive to and exit from the network) produced by any other scheduling algorithm. This is not of practical interest, since such schedules are not typically known in advance, but it offers a theoretically rigorous definition of universality that (as we shall see) helps illuminate its fundamental limits (i.e., which scheduling algorithms have the flexibility to serve as a UPS, and why).

From a more practical perspective, we say a packet scheduling algorithm is universal if it can achieve different desired performance objectives (such as fairness, reducing tail latency, minimizing flow completion times). In particular, we require that the UPS should match the performance of the best known scheduling algorithm for a given performance objective. [1]

The notion of universality for packet scheduling might seem esoteric, but we think it helps clarify some basic questions. If there exists no UPS then we should *expect* to design new scheduling algorithms as performance objectives evolve. Moreover, this would make a strong argument for switches being equipped with programmable packet schedulers so that such algorithms could be more easily deployed (as argued in [33]; in fact, it was the eloquent argument in this paper that caused us to initially ask the question about universality).

However, if there is indeed a UPS, then it changes the lens through which we view the design and evaluation of packet scheduling algorithms: e.g., rather than asking whether a new scheduling algorithm meets a performance objective, we should ask whether it is easier/cheaper to

[1] For this definition of universality, we allow the header initialization to depend on the objective being optimized. That is, while the basic scheduling operations must remain constant, the header initialization can depend on whether you are seeking fairness or minimal flow completion time.

1

Is there a universal packet scheduler?

[NSDI '16]

"You can't have *everything* you want,

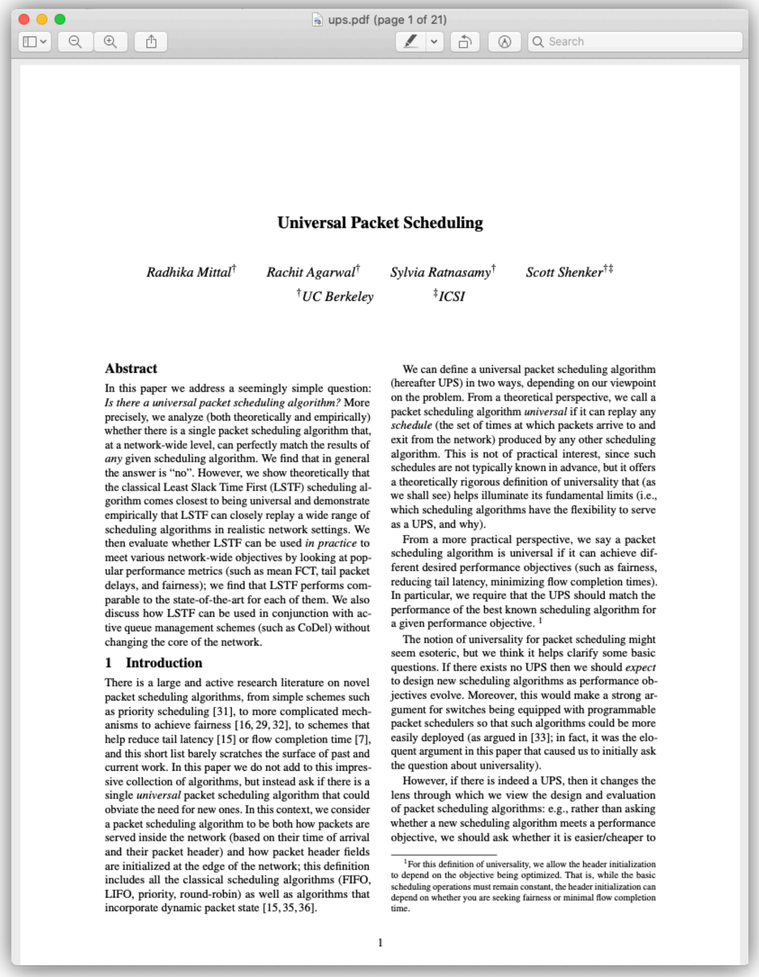
but you can have *anything* you want"

Generality

Universal packet scheduler

Flexibility

Customized algorithms



ups.pdf (page 1 of 21)

Search

Universal Packet Scheduling

Radhika Mittal[†] *Rachit Agarwal*[†] *Sylvia Ratnasamy*[†] *Scott Shenker*[‡‡]
[†]*UC Berkeley* [‡]*ICSI*

Abstract

In this paper we address a seemingly simple question: *Is there a universal packet scheduling algorithm?* More precisely, we analyze (both theoretically and empirically) whether there is a single packet scheduling algorithm that, at a network-wide level, can perfectly match the results of *any* given scheduling algorithm. We find that in general the answer is "no". However, we show theoretically that the classical Least Slack Time First (LSTF) scheduling algorithm comes closest to being universal and demonstrate empirically that LSTF can closely replay a wide range of scheduling algorithms in realistic network settings. We then evaluate whether LSTF can be used *in practice* to meet various network-wide objectives by looking at popular performance metrics (such as mean FCT, tail packet delays, and fairness); we find that LSTF performs comparable to the state-of-the-art for each of them. We also discuss how LSTF can be used in conjunction with active queue management schemes (such as CoDel) without changing the core of the network.

1 Introduction

There is a large and active research literature on novel packet scheduling algorithms, from simple schemes such as priority scheduling [31], to more complicated mechanisms to achieve fairness [16, 29, 32], to schemes that help reduce tail latency [15] or flow completion time [7], and this short list barely scratches the surface of past and current work. In this paper we do not add to this impressive collection of algorithms, but instead ask if there is a single *universal* packet scheduling algorithm that could obviate the need for new ones. In this context, we consider a packet scheduling algorithm to be both how packets are served inside the network (based on their time of arrival and their packet header) and how packet header fields are initialized at the edge of the network; this definition includes all the classical scheduling algorithms (FIFO, LIFO, priority, round-robin) as well as algorithms that incorporate dynamic packet state [15, 35, 36].

We can define a universal packet scheduling algorithm (hereafter UPS) in two ways, depending on our viewpoint on the problem. From a theoretical perspective, we call a packet scheduling algorithm *universal* if it can replay any *schedule* (the set of times at which packets arrive to and exit from the network) produced by any other scheduling algorithm. This is not of practical interest, since such schedules are not typically known in advance, but it offers a theoretically rigorous definition of universality that (as we shall see) helps illuminate its fundamental limits (i.e., which scheduling algorithms have the flexibility to serve as a UPS, and why).

From a more practical perspective, we say a packet scheduling algorithm is universal if it can achieve different desired performance objectives (such as fairness, reducing tail latency, minimizing flow completion times). In particular, we require that the UPS should match the performance of the best known scheduling algorithm for a given performance objective. [1]

The notion of universality for packet scheduling might seem esoteric, but we think it helps clarify some basic questions. If there exists no UPS then we should *expect* to design new scheduling algorithms as performance objectives evolve. Moreover, this would make a strong argument for switches being equipped with programmable packet schedulers so that such algorithms could be more easily deployed (as argued in [33]; in fact, it was the eloquent argument in this paper that caused us to initially ask the question about universality).

However, if there is indeed a UPS, then it changes the lens through which we view the design and evaluation of packet scheduling algorithms: e.g., rather than asking whether a new scheduling algorithm meets a performance objective, we should ask whether it is easier/cheaper to

[1] For this definition of universality, we allow the header initialization to depend on the objective being optimized. That is, while the basic scheduling operations must remain constant, the header initialization can depend on whether you are seeking fairness or minimal flow completion time.

1

Is there a universal packet scheduler?

[NSDI '16]

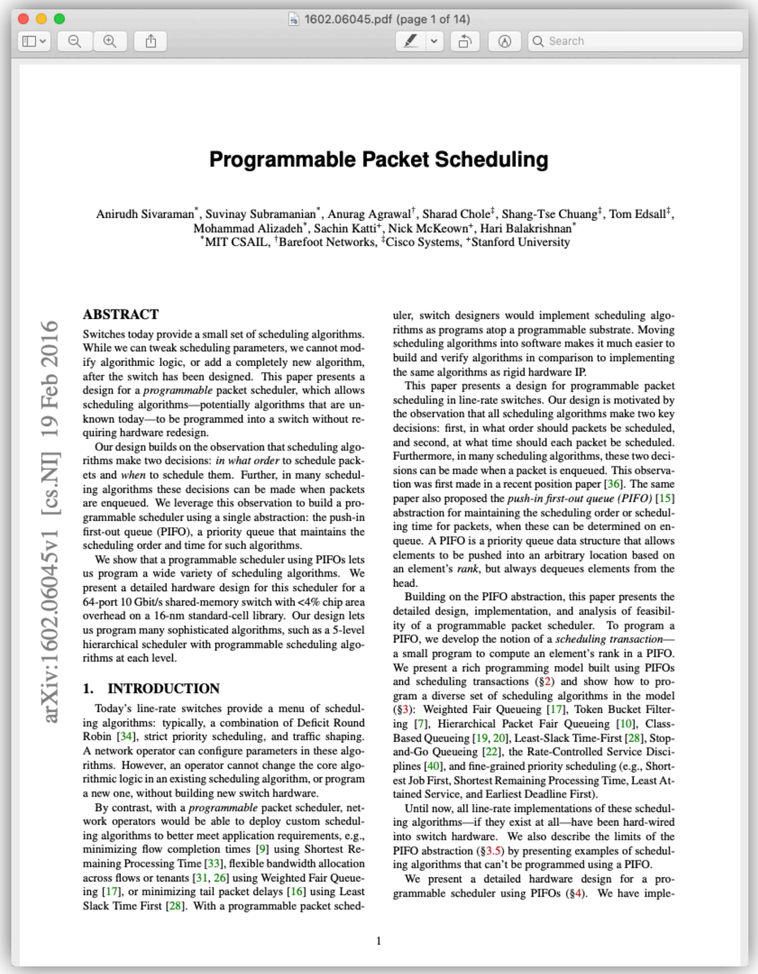
"You can't have *everything* you want,

but you can have *anything* you want"

Generality

Universal packet scheduler

Programmable scheduling



1602.06045.pdf (page 1 of 14)

Search

Programmable Packet Scheduling

Anirudh Sivaraman*, Suvinay Subramanian*, Anurag Agrawal†, Sharad Chole‡, Shang-Tse Chuang‡, Tom Edsall‡, Mohammad Alizadeh*, Sachin Katti+, Nick McKeown+, Hari Balakrishnan*
*MIT CSAIL, †Barefoot Networks, ‡Cisco Systems, +Stanford University

ABSTRACT

Switches today provide a small set of scheduling algorithms. While we can tweak scheduling parameters, we cannot modify algorithmic logic, or add a completely new algorithm, after the switch has been designed. This paper presents a design for a *programmable* packet scheduler, which allows scheduling algorithms—potentially algorithms that are unknown today—to be programmed into a switch without requiring hardware redesign.

Our design builds on the observation that scheduling algorithms make two decisions: *in what order* to schedule packets and *when* to schedule them. Further, in many scheduling algorithms these decisions can be made when packets are enqueued. We leverage this observation to build a programmable scheduler using a single abstraction: the push-in first-out queue (PIFO), a priority queue that maintains the scheduling order and time for such algorithms.

We show that a programmable scheduler using PIFOs lets us program a wide variety of scheduling algorithms. We present a detailed hardware design for this scheduler for a 64-port 10 Gbit/s shared-memory switch with <4% chip area overhead on a 16-nm standard-cell library. Our design lets us program many sophisticated algorithms, such as a 5-level hierarchical scheduler with programmable scheduling algorithms at each level.

1. INTRODUCTION

Today's line-rate switches provide a menu of scheduling algorithms: typically, a combination of Deficit Round Robin [34], strict priority scheduling, and traffic shaping. A network operator can configure parameters in these algorithms. However, an operator cannot change the core algorithmic logic in an existing scheduling algorithm, or program a new one, without building new switch hardware.

By contrast, with a *programmable* packet scheduler, network operators would be able to deploy custom scheduling algorithms to better meet application requirements, e.g., minimizing flow completion times [9] using Shortest Remaining Processing Time [33], flexible bandwidth allocation across flows or tenants [31, 26] using Weighted Fair Queueing [17], or minimizing tail packet delays [16] using Least Slack Time First [28]. With a programmable packet scheduler, switch designers would implement scheduling algorithms as programs atop a programmable substrate. Moving scheduling algorithms into software makes it much easier to build and verify algorithms in comparison to implementing the same algorithms as rigid hardware IP.

This paper presents a design for programmable packet scheduling in line-rate switches. Our design is motivated by the observation that all scheduling algorithms make two key decisions: first, in what order should packets be scheduled, and second, at what time should each packet be scheduled. Furthermore, in many scheduling algorithms, these two decisions can be made when a packet is enqueued. This observation was first made in a recent position paper [36]. The same paper also proposed the *push-in first-out queue (PIFO)* [15] abstraction for maintaining the scheduling order or scheduling time for packets, when these can be determined on enqueue. A PIFO is a priority queue data structure that allows elements to be pushed into an arbitrary location based on an element's *rank*, but always dequeues elements from the head.

Building on the PIFO abstraction, this paper presents the detailed design, implementation, and analysis of feasibility of a programmable packet scheduler. To program a PIFO, we develop the notion of a *scheduling transaction*—a small program to compute an element's rank in a PIFO. We present a rich programming model built using PIFOs and scheduling transactions (§2) and show how to program a diverse set of scheduling algorithms in the model (§3): Weighted Fair Queueing [17], Token Bucket Filtering [7], Hierarchical Packet Fair Queueing [10], Class-Based Queueing [19, 20], Least-Slack Time-First [28], Stop-and-Go Queueing [22], the Rate-Controlled Service Disciplines [40], and fine-grained priority scheduling (e.g., Shortest Job First, Shortest Remaining Processing Time, Least Attained Service, and Earliest Deadline First).

Until now, all line-rate implementations of these scheduling algorithms—if they exist at all—have been hard-wired into switch hardware. We also describe the limits of the PIFO abstraction (§3.5) by presenting examples of scheduling algorithms that can't be programmed using a PIFO.

We present a detailed hardware design for a programmable scheduler using PIFOs (§4). We have imple-

1

PIFO abstraction for programmable scheduling

[SIGCOMM '16]

Implementing PIFO queues in hardware is difficult

Deployability	Requires new ASIC implementation, which might take years
Scalability	Supports ~1k flows and ~10 Gbps
Flexibility	Assumes monotonic increase of ranks within flows

Can we approximate PIFO queues…

at line rate,

at scale, and

on existing devices?

Introducing…

SP-PIFO

A *deployable, scalable and flexible*

PIFO approximation

PIFO queues can be used as an abstraction to make scheduling programmable

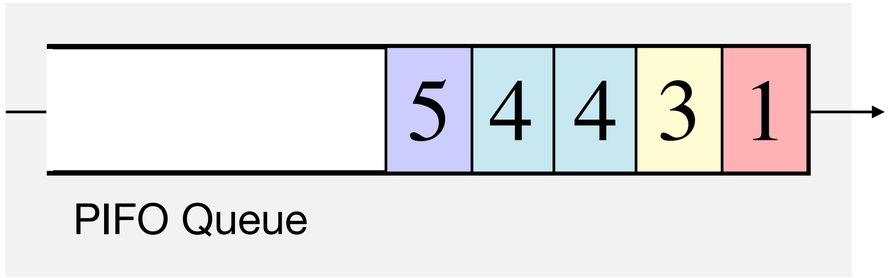
The PIFO queue

Allows packets to be pushed into arbitrary locations

Only drains packets from the head

Packet ranks

Input sequence



Output sequence

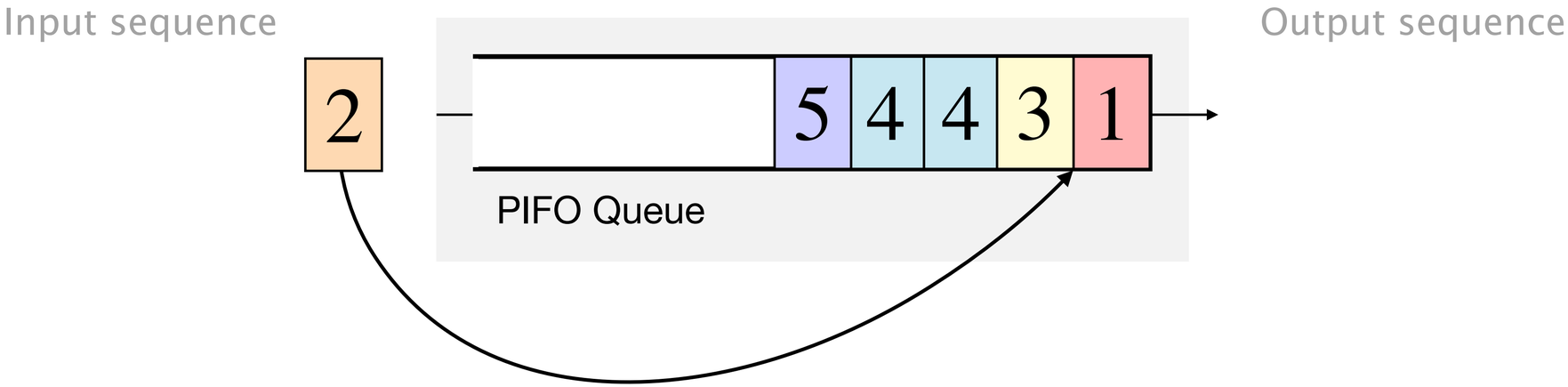
PIFO queues can be used as an abstraction to make scheduling programmable

The PIFO queue

Allows packets to be pushed into arbitrary locations

Only drains packets from the head

Packet ranks



PIFO queues can be used as an abstraction to make scheduling programmable

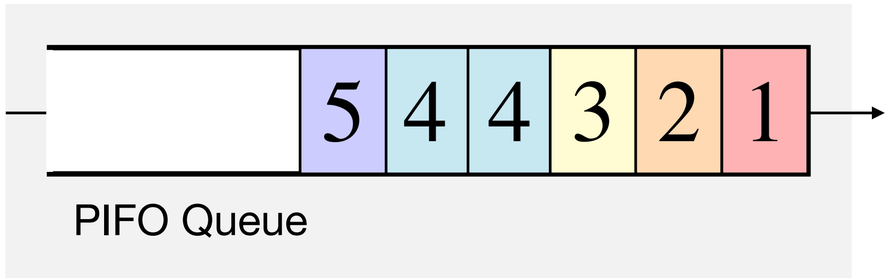
The PIFO queue

Allows packets to be pushed into arbitrary locations

Only drains packets from the head

Packet ranks

Input sequence



Output sequence

PIFO queues can be used as an abstraction to make scheduling programmable

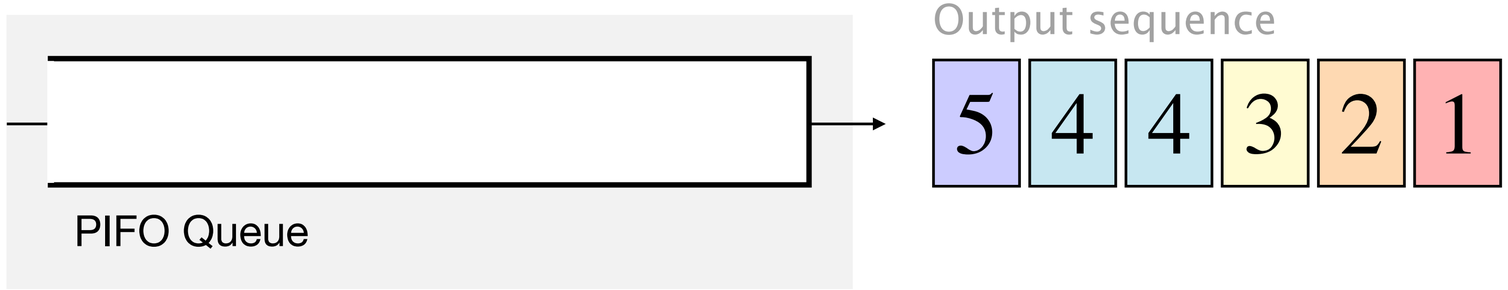
The PIFO queue

Allows packets to be pushed into arbitrary locations

Only drains packets from the head

Packet ranks

Input sequence



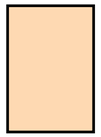
PIFO queues can be used as an abstraction to make scheduling programmable

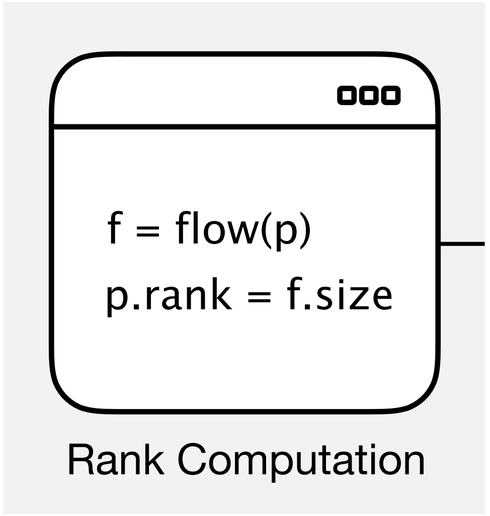
Programmable scheduler

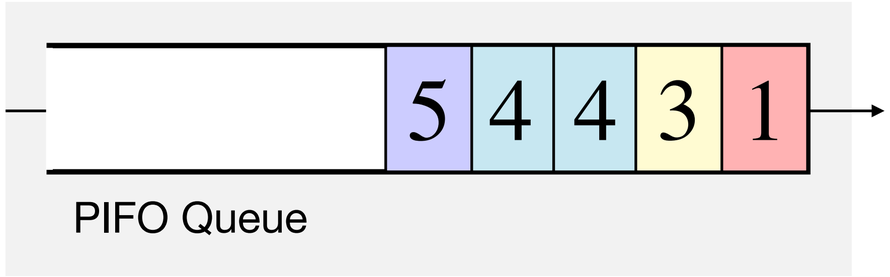
Rank computation (programmable)

PIFO queue (fixed logic)

Input sequence







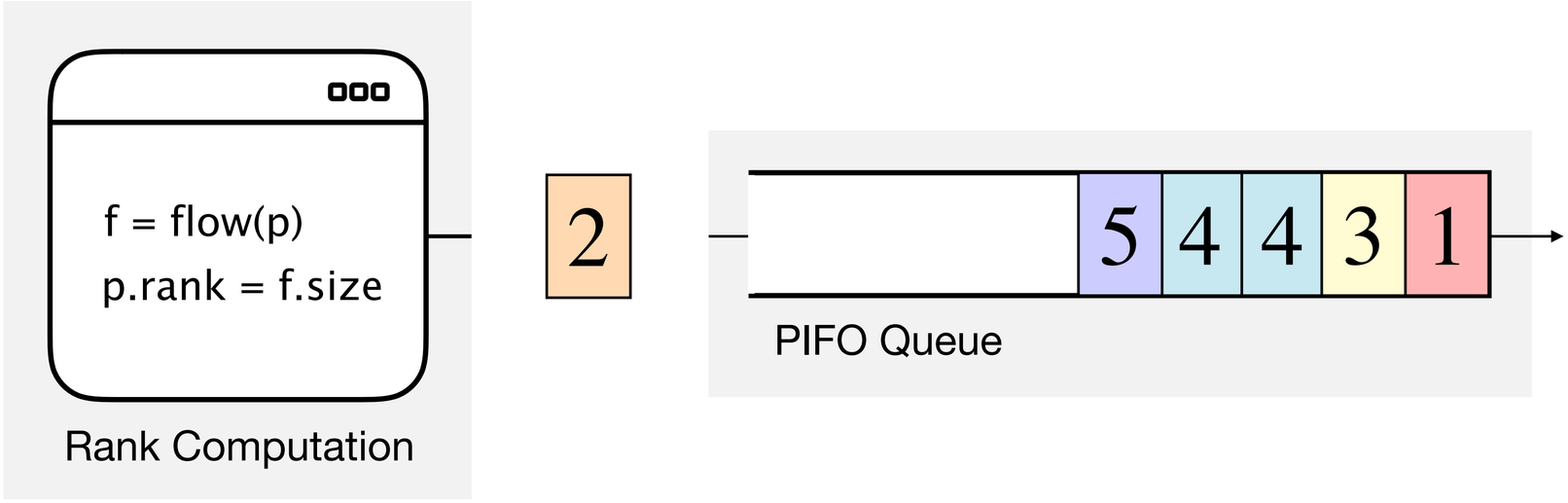
PIFO queues can be used as an abstraction to make scheduling programmable

Programmable scheduler

Rank computation (programmable)

PIFO queue (fixed logic)

Input sequence



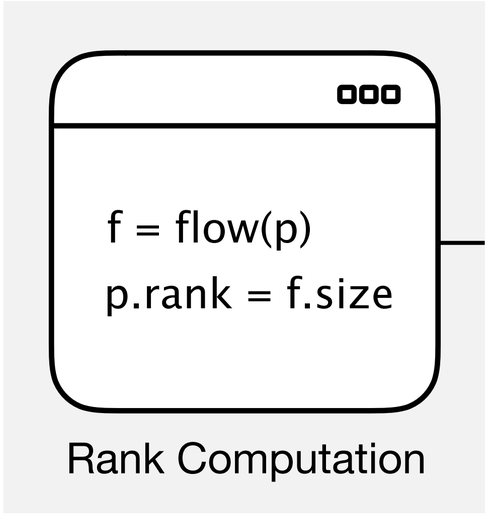
PIFO queues can be used as an abstraction to make scheduling programmable

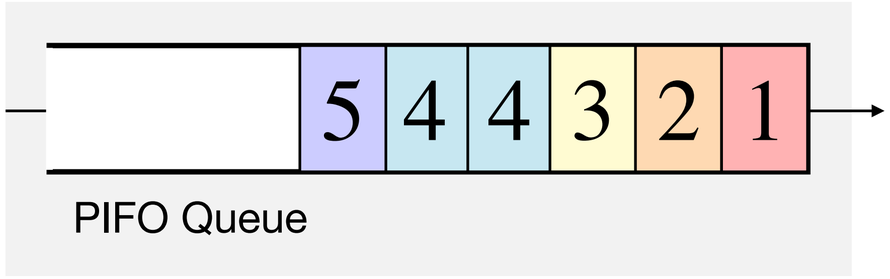
Programmable scheduler

Rank computation (programmable)

PIFO queue (fixed logic)

Input sequence



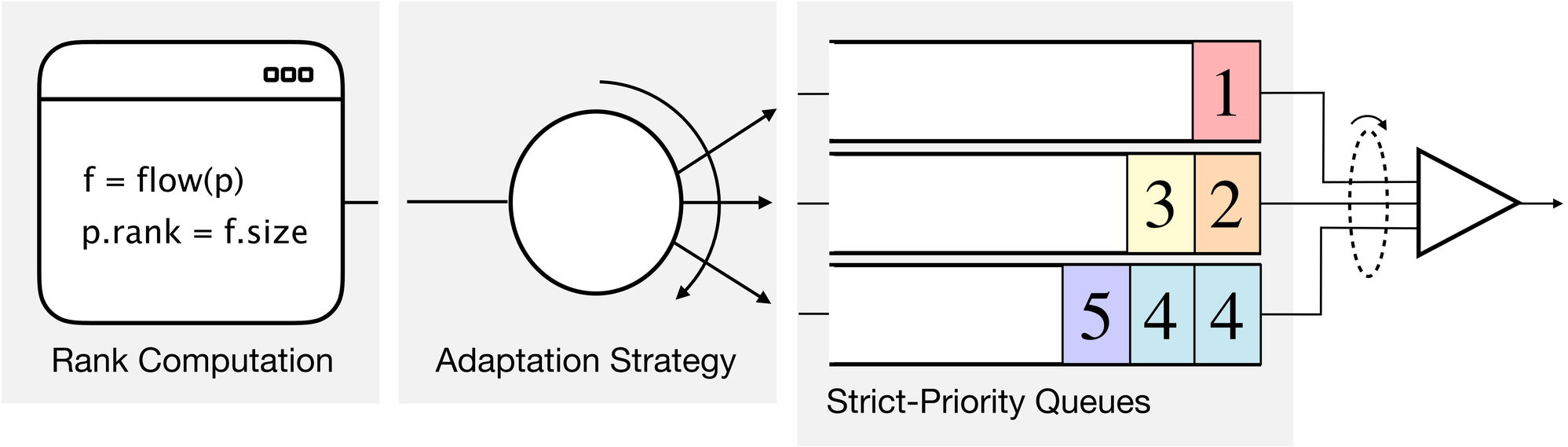


SP-PIFO approximates PIFO behaviors using Strict-Priority queues and a dynamic mapping strategy

Programmable scheduler

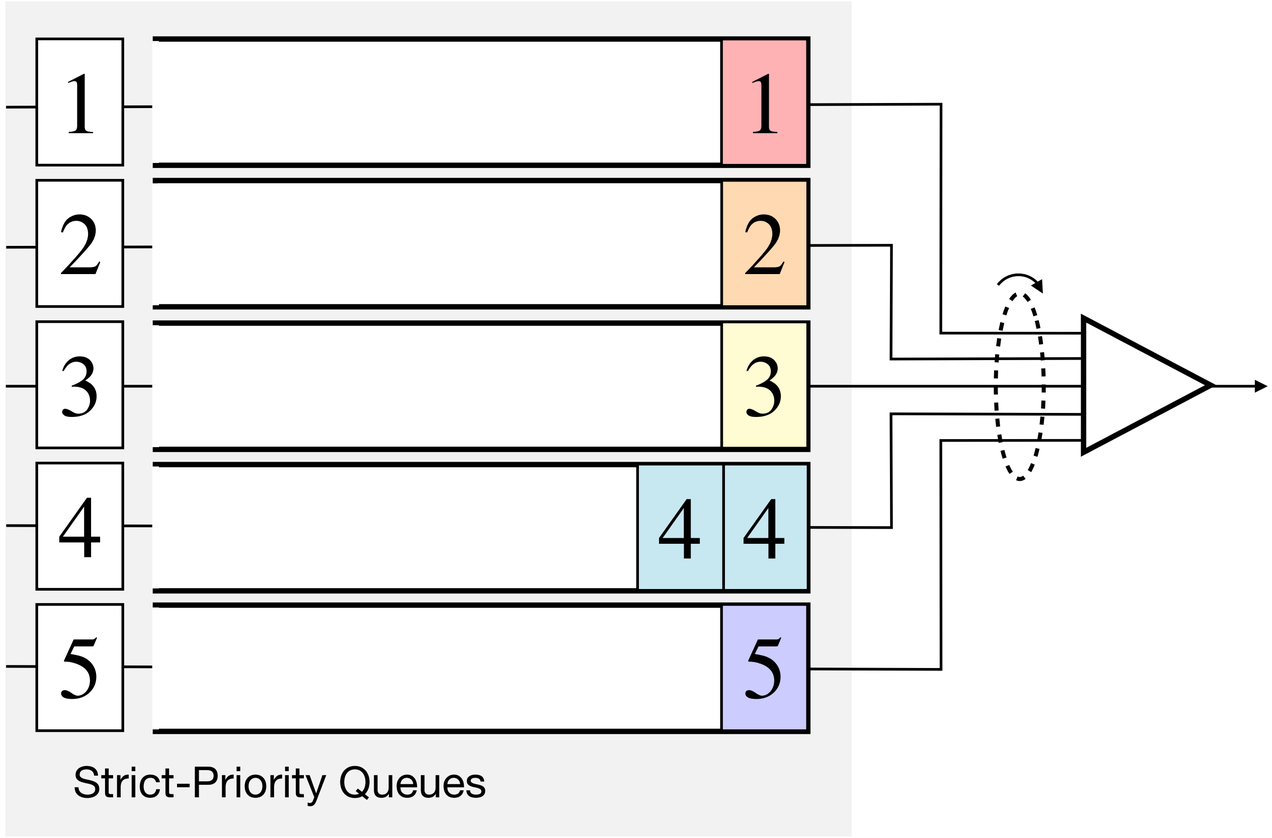
Rank computation (programmable)

Adaptation strategy + strict-priority queues (fixed logic)



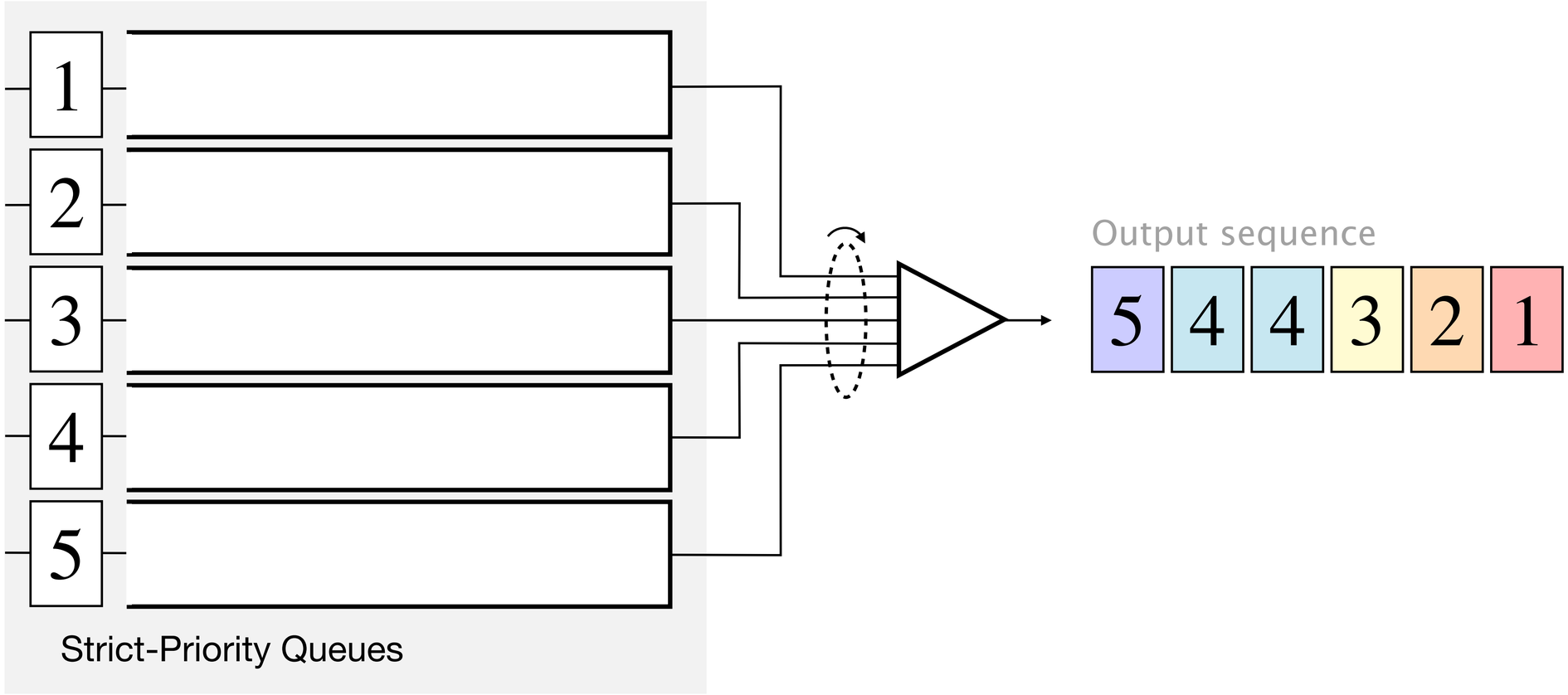
SP-PIFO approximates PIFO behaviors using Strict-Priority queues and a dynamic mapping strategy

Ideal case Perfect PIFO if number of queues >= number of ranks



SP-PIFO approximates PIFO behaviors using Strict-Priority queues and a dynamic mapping strategy

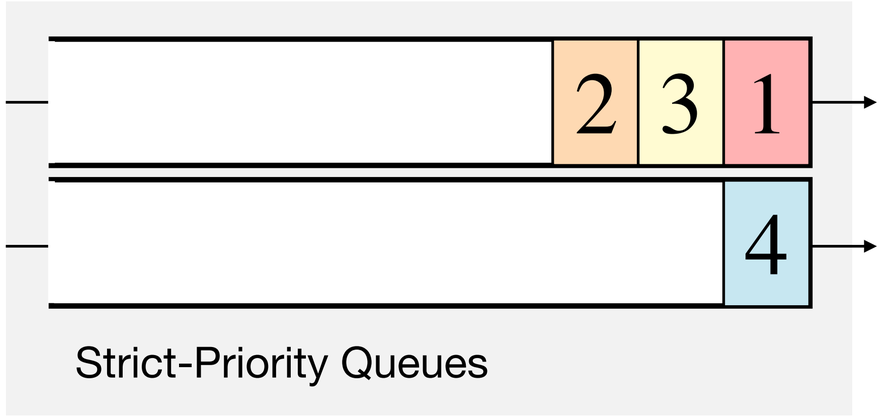
Ideal case Perfect PIFO if number of queues >= number of ranks



SP-PIFO approximates PIFO behaviors using Strict-Priority queues and a dynamic mapping strategy

In practice Number of queues < number of ranks

Problem Output sequence can have scheduling errors

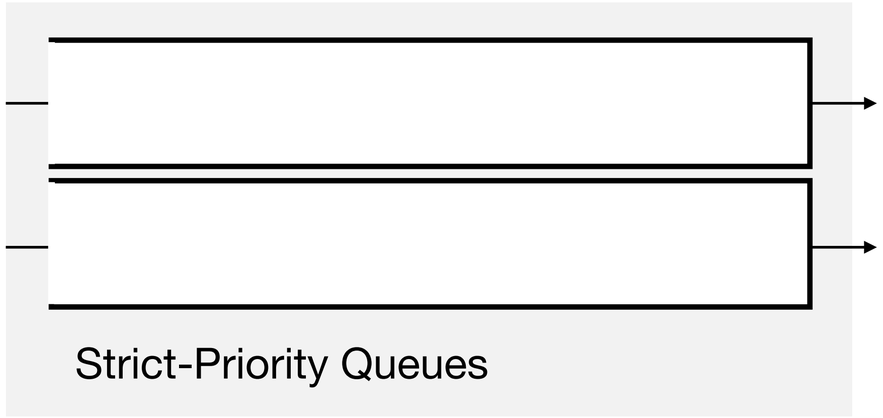


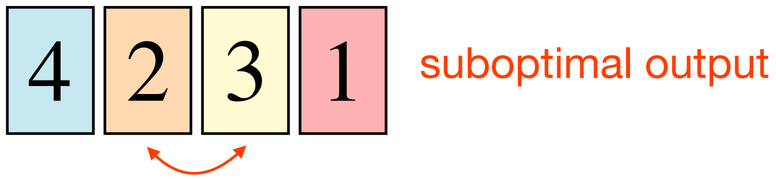
SP-PIFO approximates PIFO behaviors using Strict-Priority queues and a dynamic mapping strategy

In practice Number of queues < number of ranks

Problem Output sequence can have scheduling errors

Low-ranked packets drained after high-ranked packets





suboptimal output

SP-PIFO approximates PIFO behaviors using Strict-Priority queues and a dynamic mapping strategy

In practice Number of queues < number of ranks

Problem Output sequence can have scheduling errors

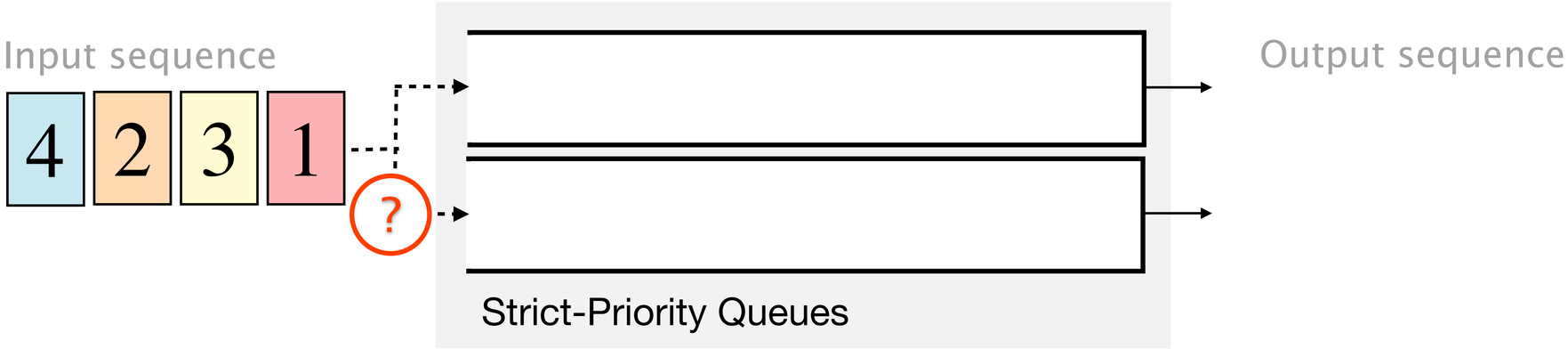
Opportunity Design mapping strategies that minimize scheduling errors

SP-PIFO defines mapping through ‘queue bounds'

Mapping

Queue bounds scanned bottom-up

Packet enqueued if rank >= queue bound

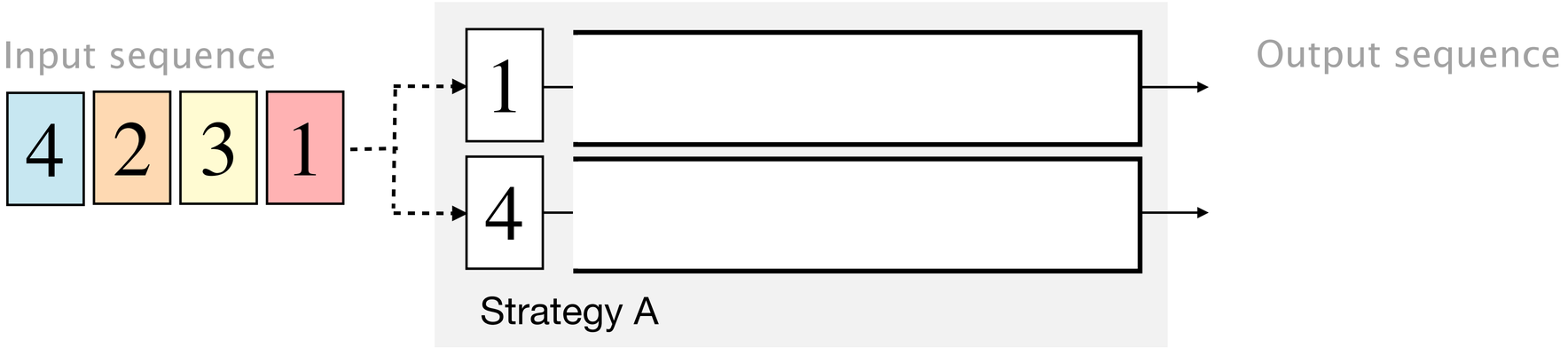


SP-PIFO defines mapping through ‘queue bounds'

Mapping

Queue bounds scanned bottom-up

Packet enqueued if rank >= queue bound

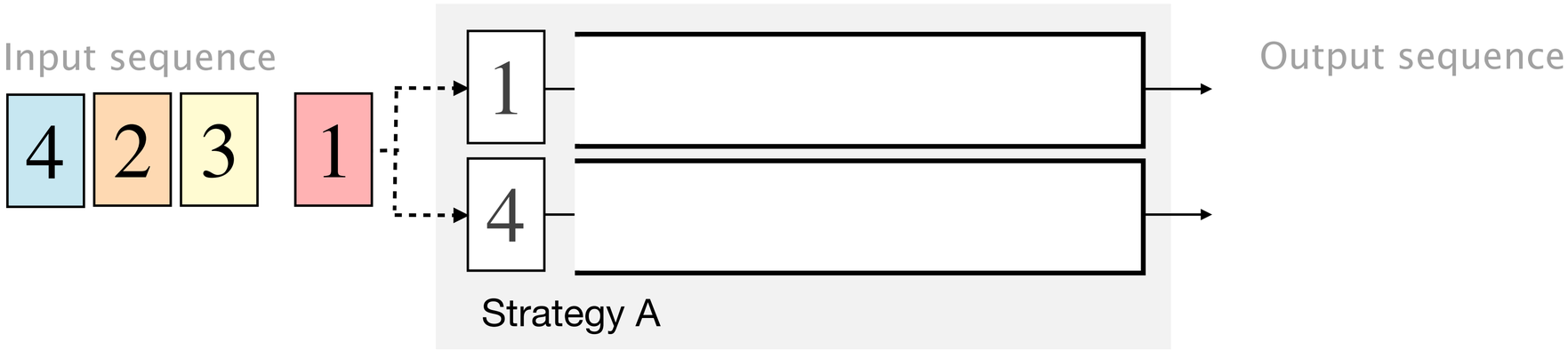


SP-PIFO defines mapping through ‘queue bounds'

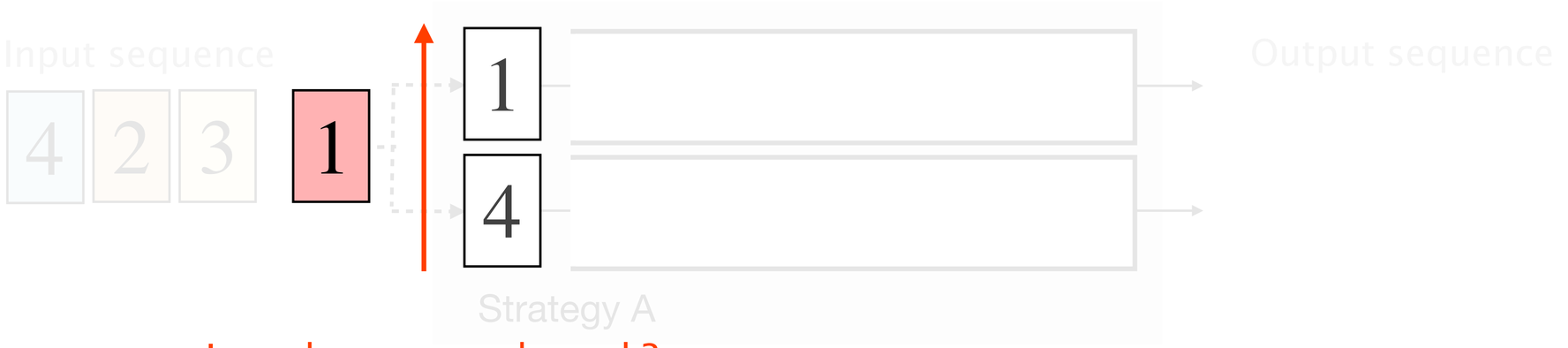
Mapping

Queue bounds scanned bottom-up

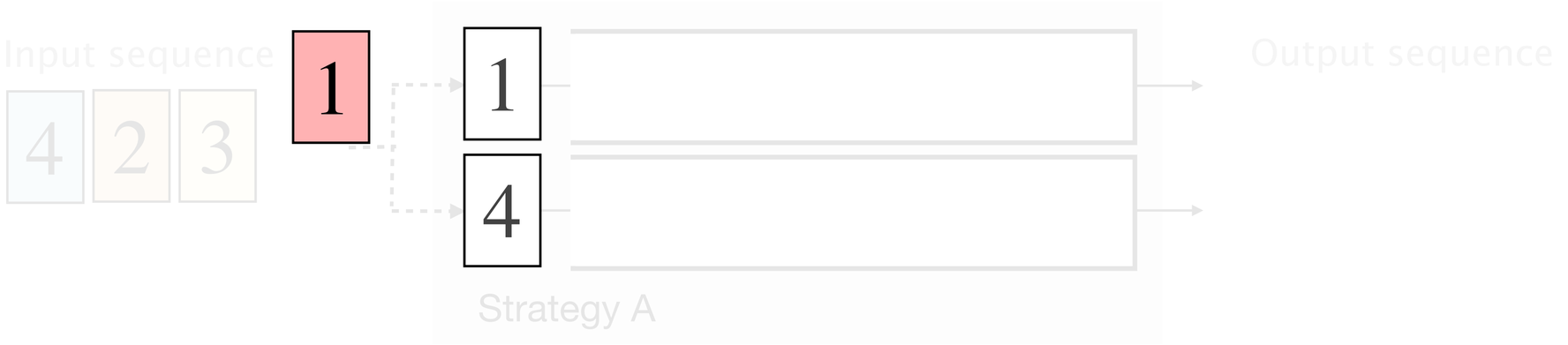
Packet enqueued if rank >= queue bound



SP-PIFO defines mapping through ‘queue bounds'



SP-PIFO defines mapping through ‘queue bounds'

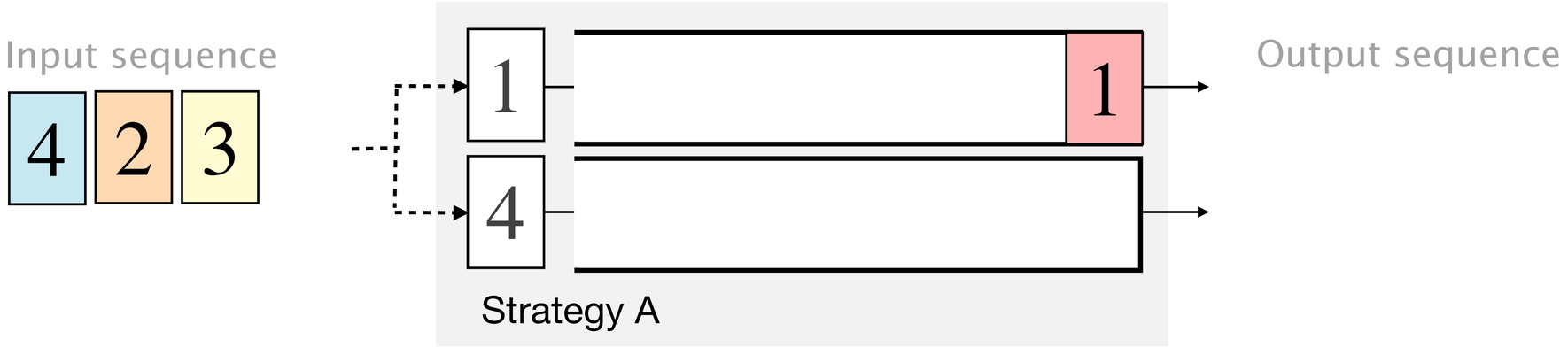


SP-PIFO defines mapping through ‘queue bounds'

Mapping

Queue bounds scanned bottom-up

Packet enqueued if rank >= queue bound



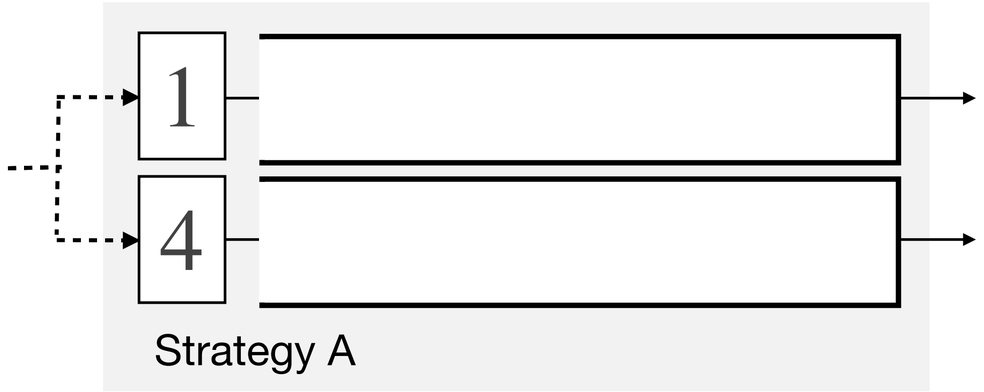
SP-PIFO defines mapping through ‘queue bounds'

Mapping

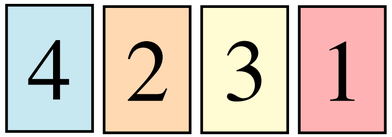
Queue bounds scanned bottom-up

Packet enqueued if rank >= queue bound

Input sequence



Output sequence



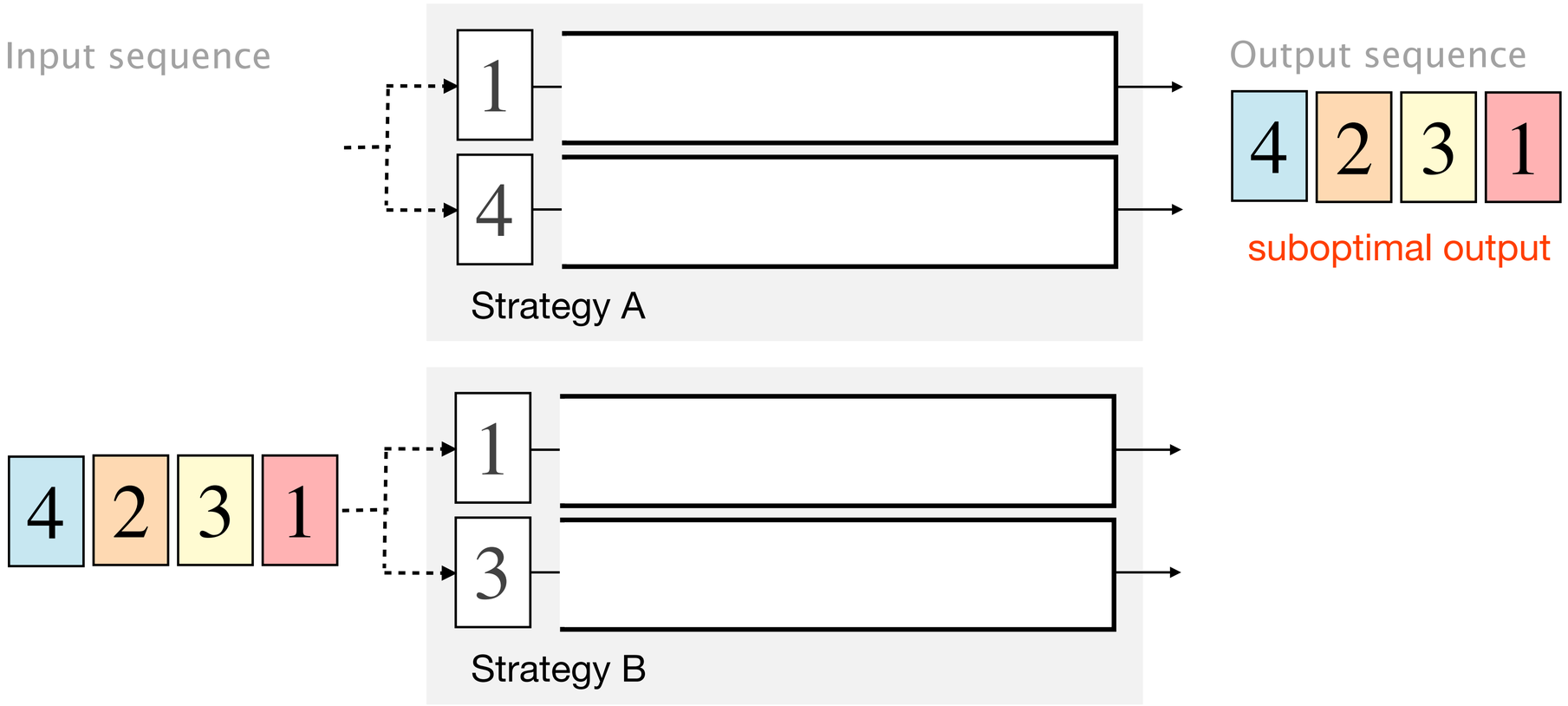
suboptimal output

SP-PIFO defines mapping through ‘queue bounds'

Mapping

Queue bounds scanned bottom-up

Packet enqueued if rank >= queue bound



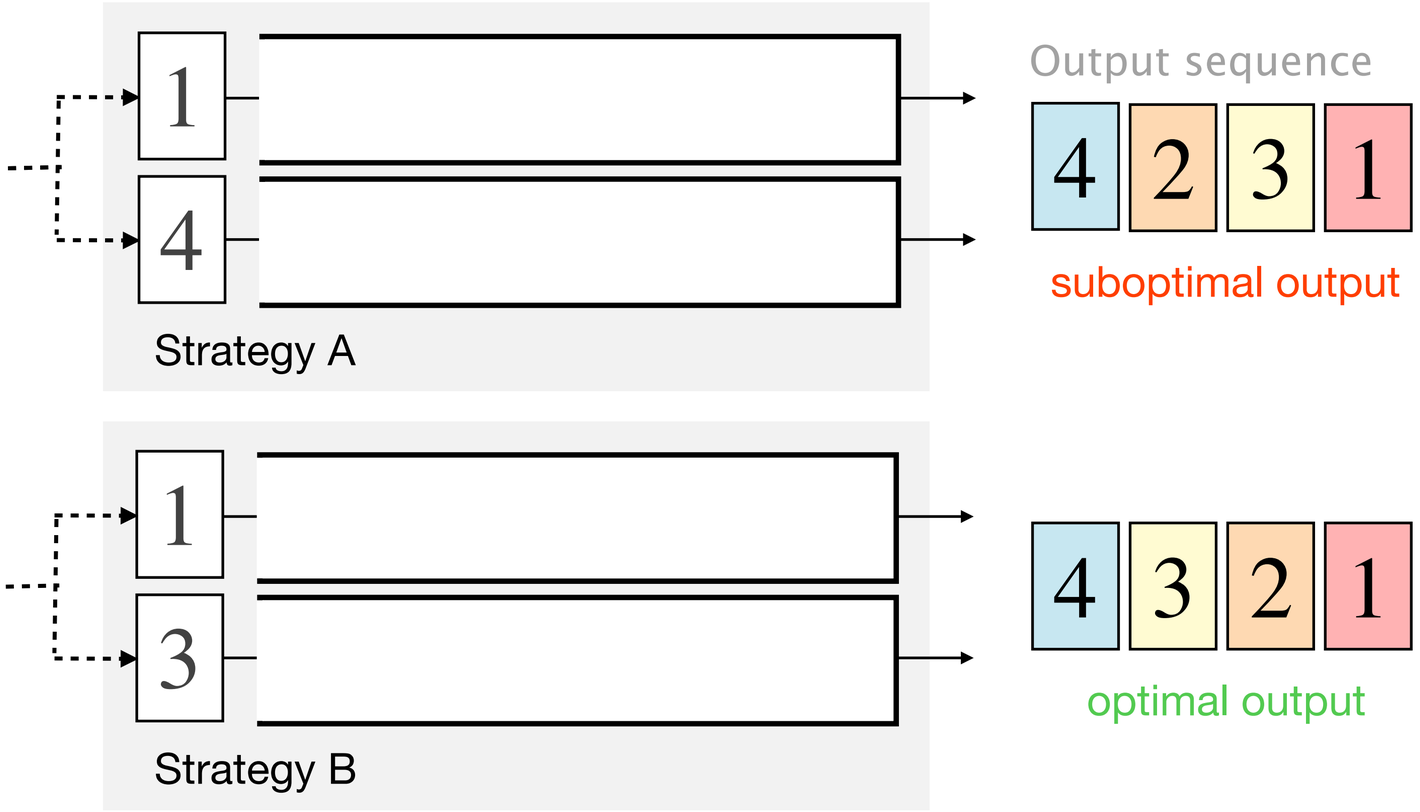
SP-PIFO defines mapping through ‘queue bounds'

Mapping

Queue bounds scanned bottom-up

Packet enqueued if rank >= queue bound

Input sequence



How can we design a mapping strategy that minimizes scheduling errors?

SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

1 Adaptation design
How does it work

2 Implementation
How can it be deployed

3 Evaluation
How well does it perform

SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

1 **Adaptation design**
How does it work

2 **Implementation**
How can it be deployed

3 **Evaluation**
How well does it perform

Problem formulation

Objective Find optimal queue bounds $\boldsymbol{q}^*$

That minimize the expected loss U for all ranks

$$\boldsymbol{q}^* = \underset{\boldsymbol{q} \in Q}{argmin}\ \underset{r \sim R}{E}\left[\ U(\boldsymbol{q}, r)\ \right]$$

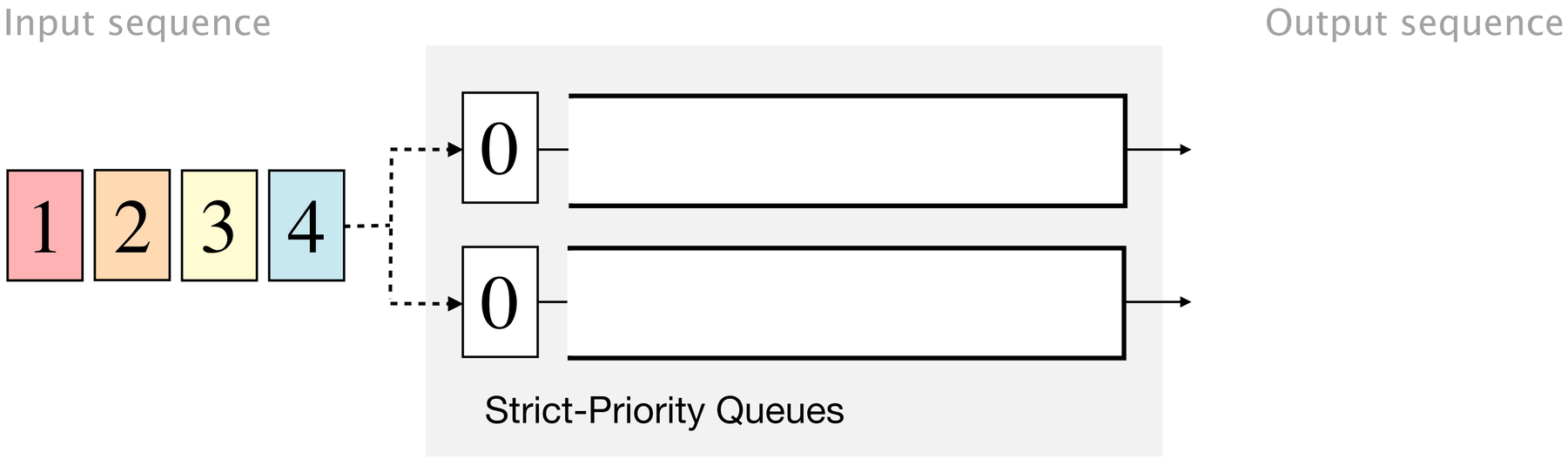
Unpifoness (U) quantifies the scheduling errors

SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Initialization

Zero traffic knowledge

Queue bounds set to zero

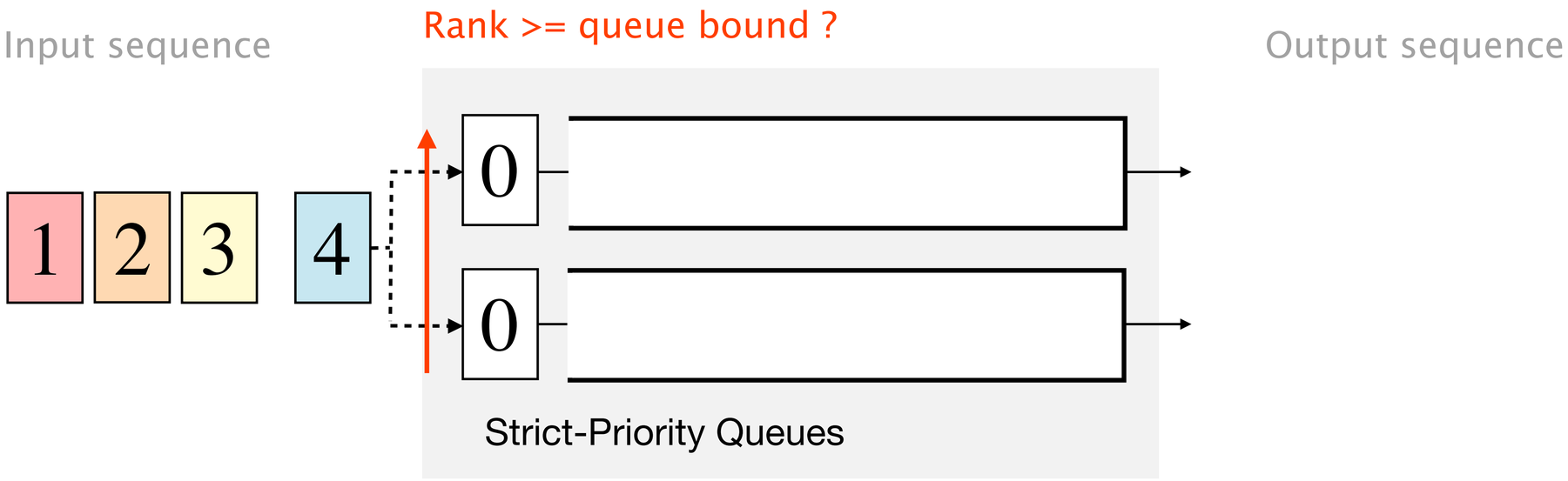


SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Initialization

Zero traffic knowledge

Queue bounds set to zero

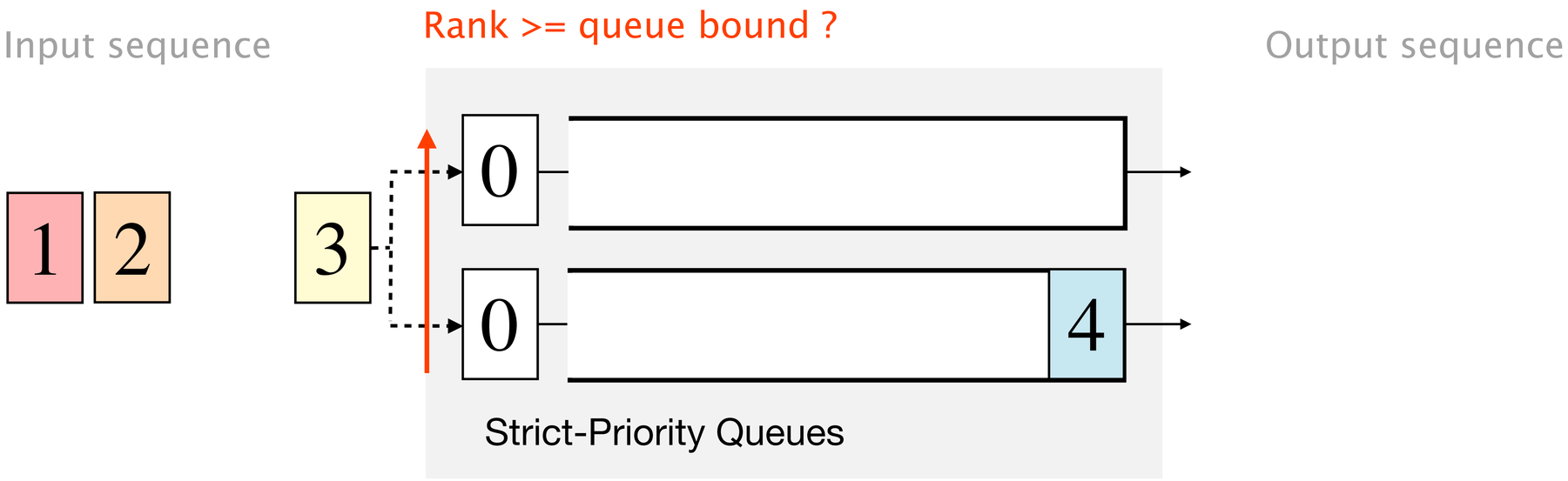


SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Initialization

Zero traffic knowledge

Queue bounds set to zero

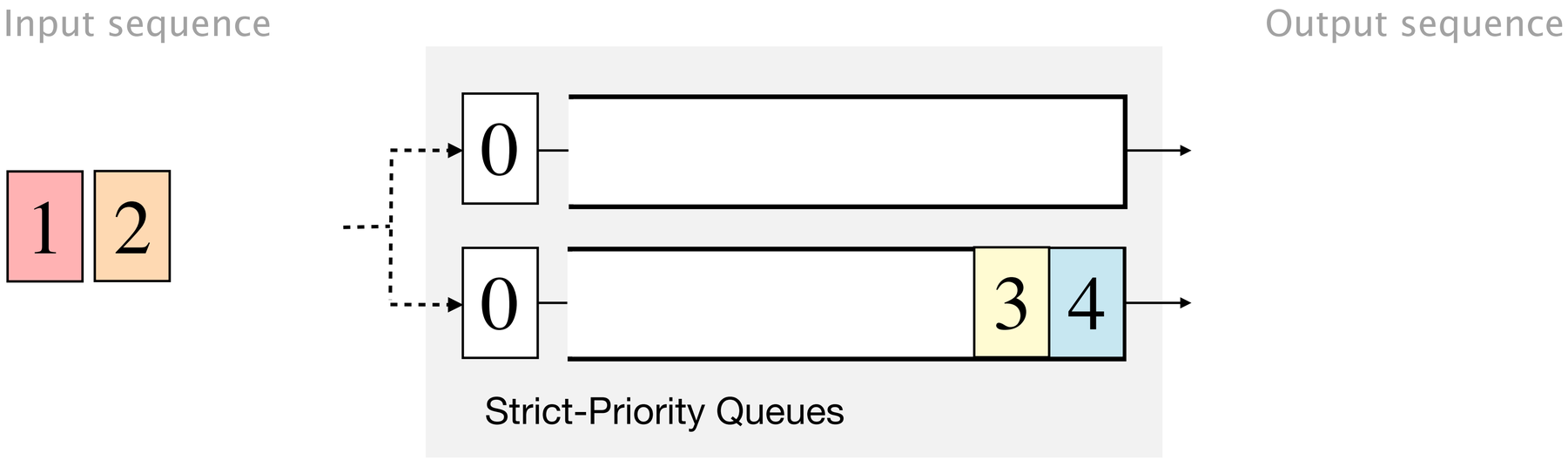


SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Initialization

Zero traffic knowledge

Queue bounds set to zero

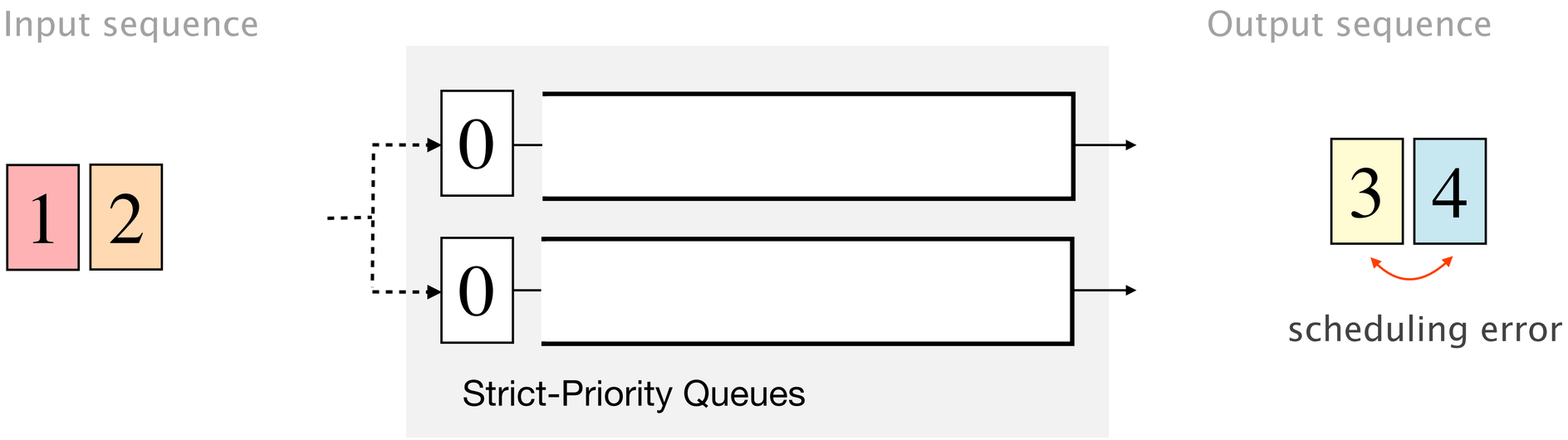


SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Initialization

Zero traffic knowledge

Queue bounds set to zero

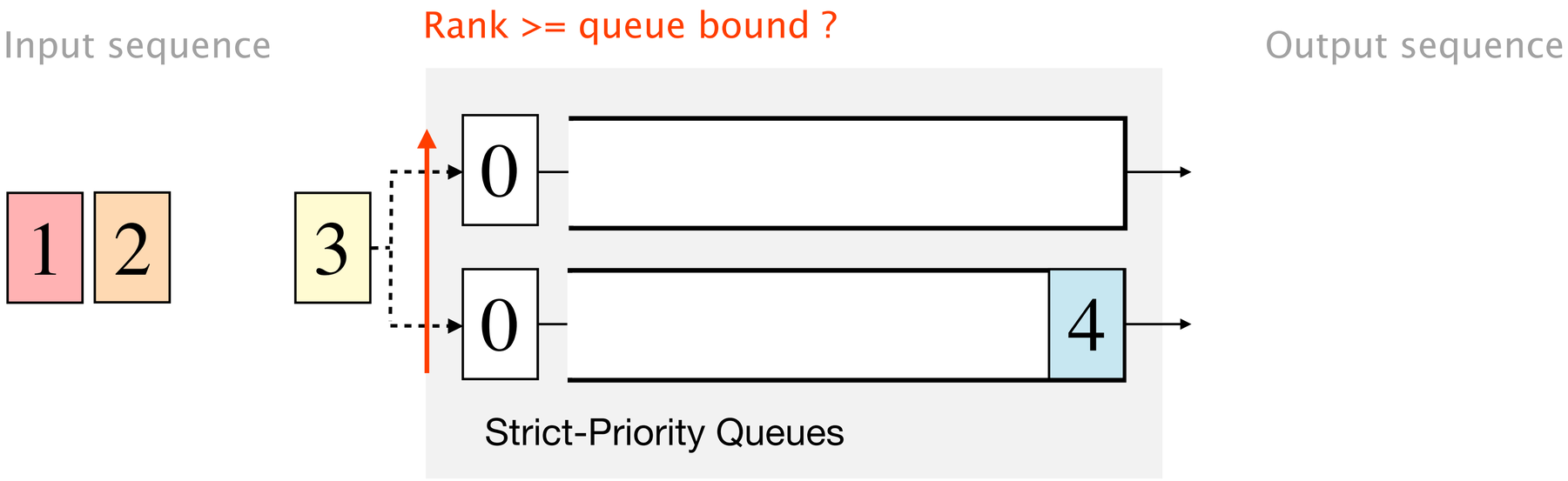


SP-PIFO adapts the mapping of packet ranks to strict-priority queues

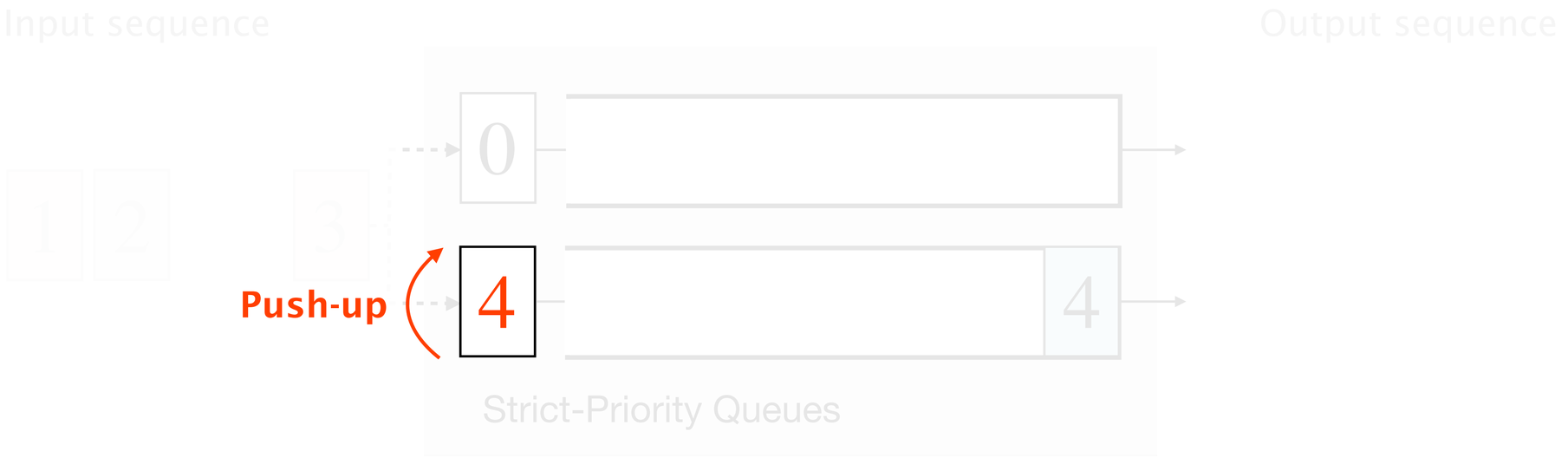
Initialization

Zero traffic knowledge

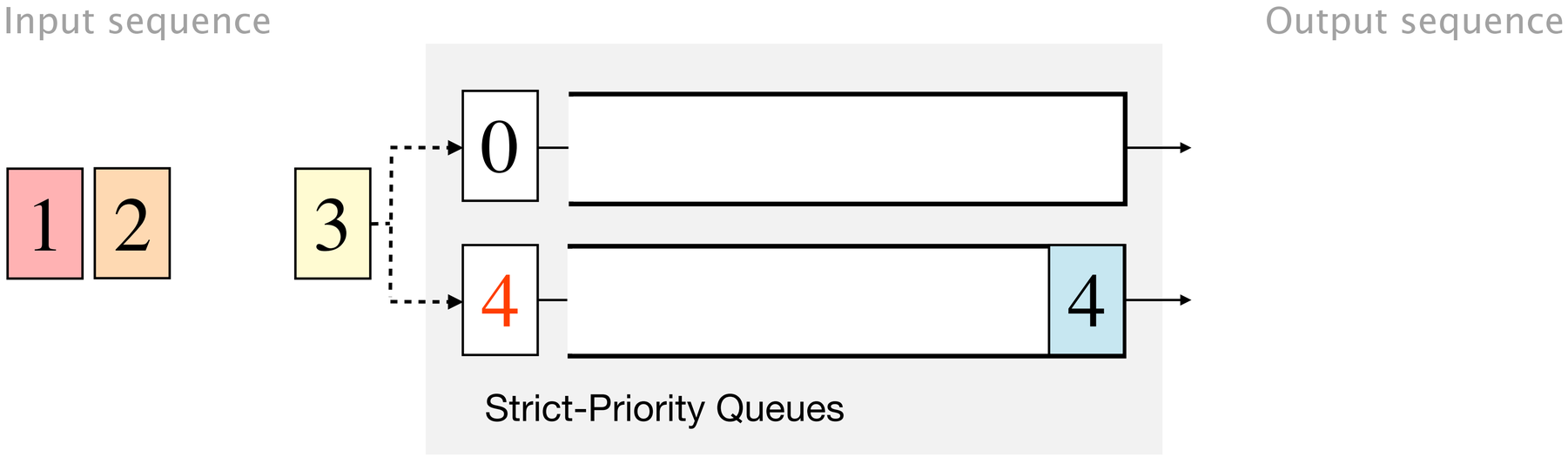
Queue bounds set to zero



SP-PIFO adapts the mapping of packet ranks to strict-priority queues



SP-PIFO adapts the mapping of packet ranks to strict-priority queues

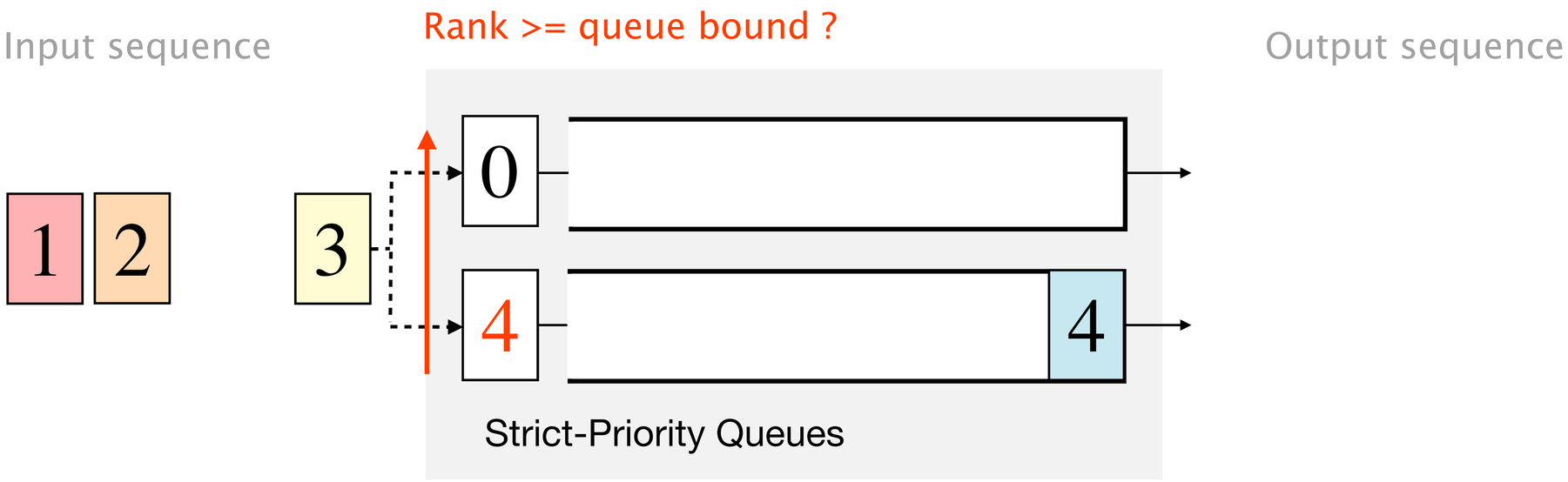


SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Push-up

(future low-rank packets to higher-priority queues)

After enqueue, queue bound set to the rank of the packet enqueued

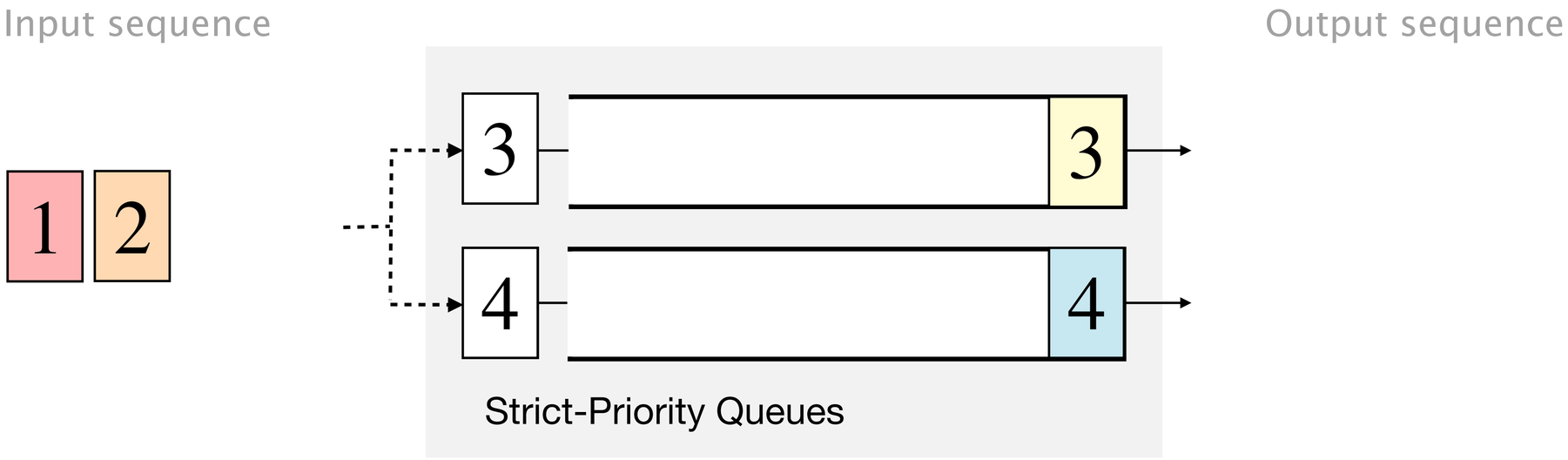


SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Push-up
(future low-rank packets to higher-priority queues)

After enqueue, queue bound set to the rank of the packet enqueued

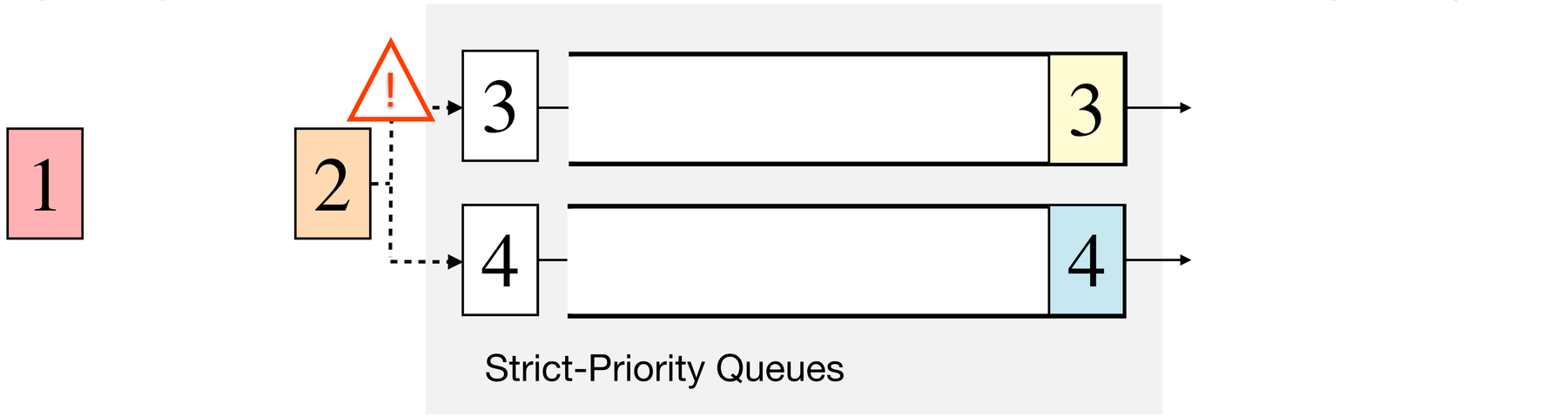
SP-PIFO adapts the mapping of packet ranks to strict-priority queues



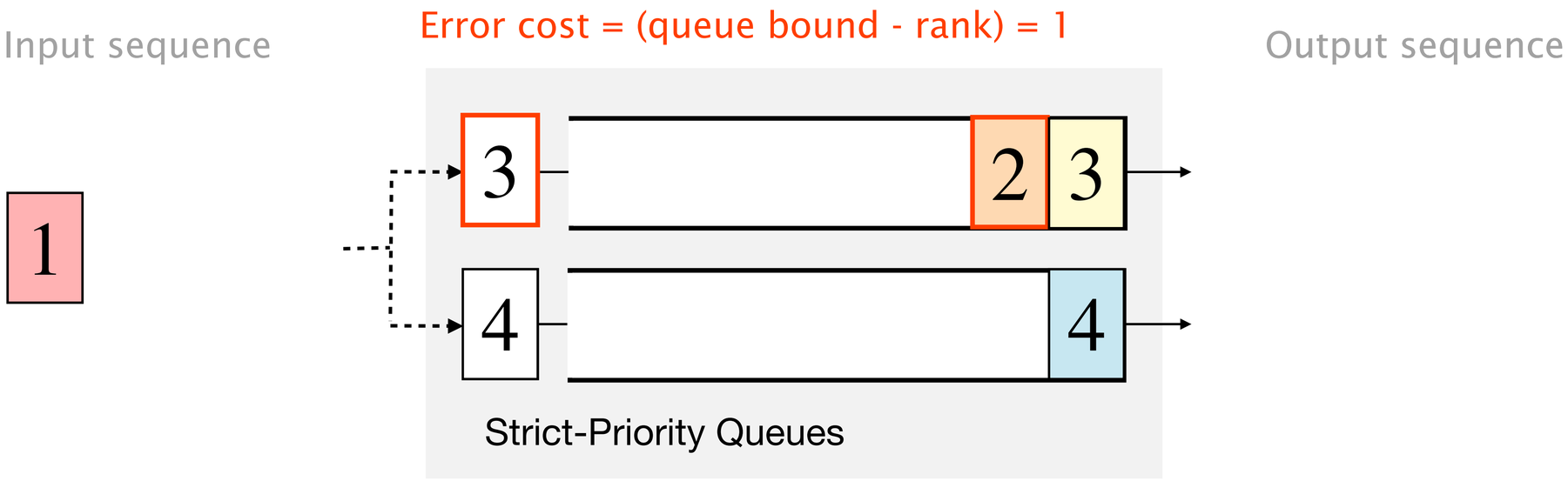
SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Push-up
(future low-rank packets to higher-priority queues)

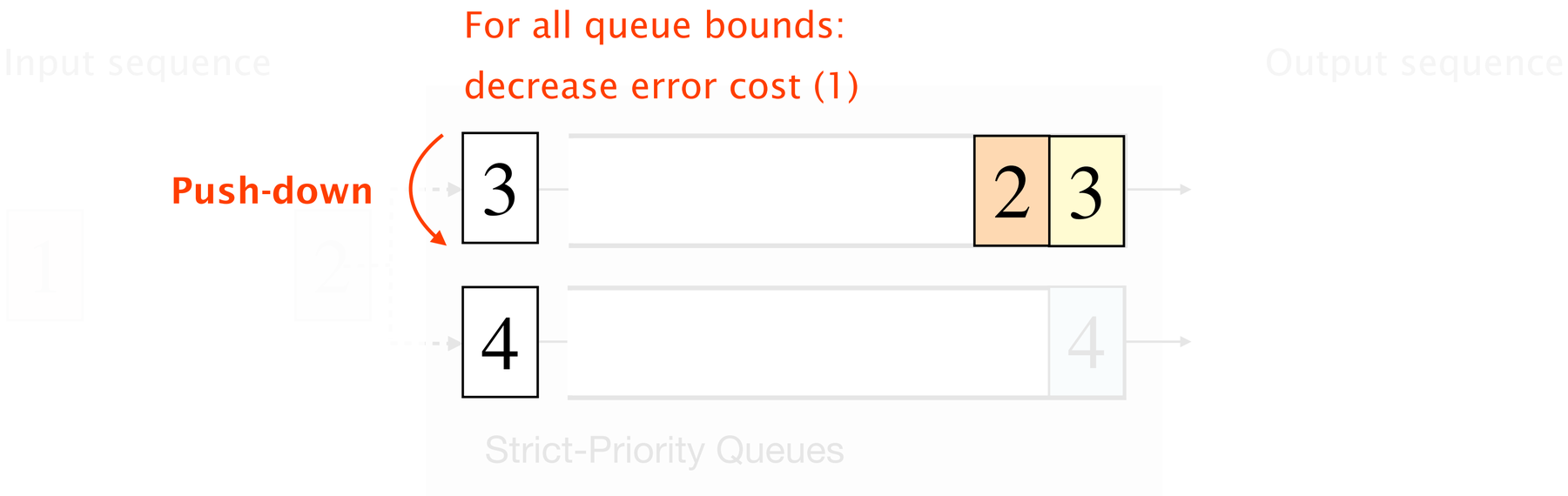
After enqueue, queue bound set to the rank of the packet enqueued



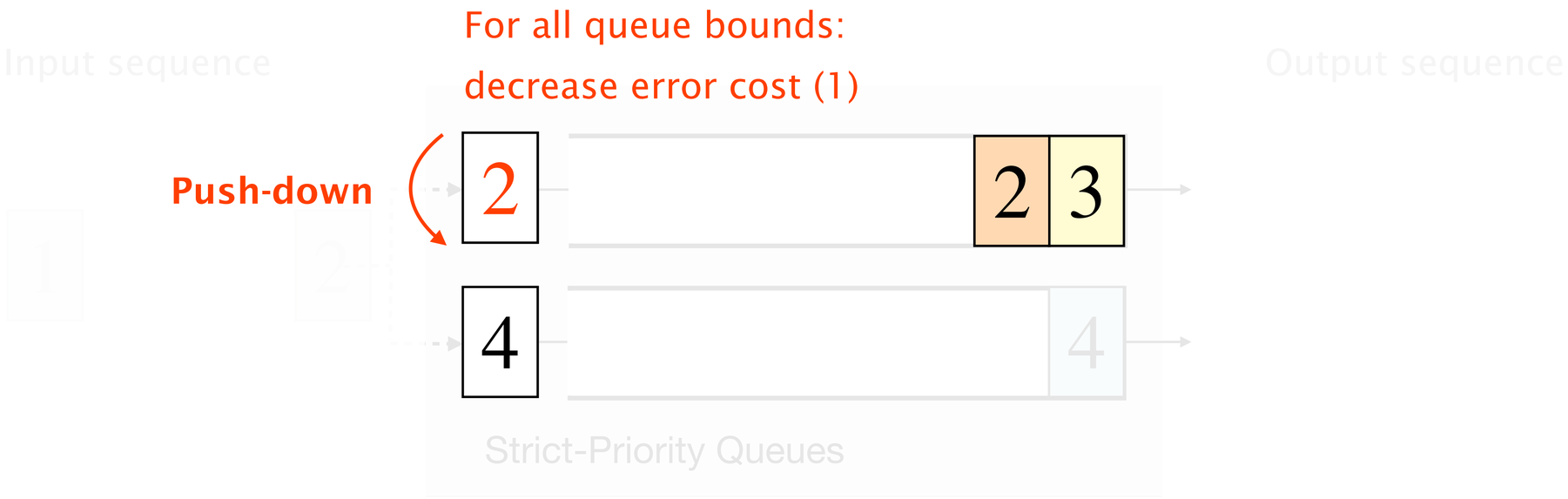
SP-PIFO adapts the mapping of packet ranks to strict-priority queues



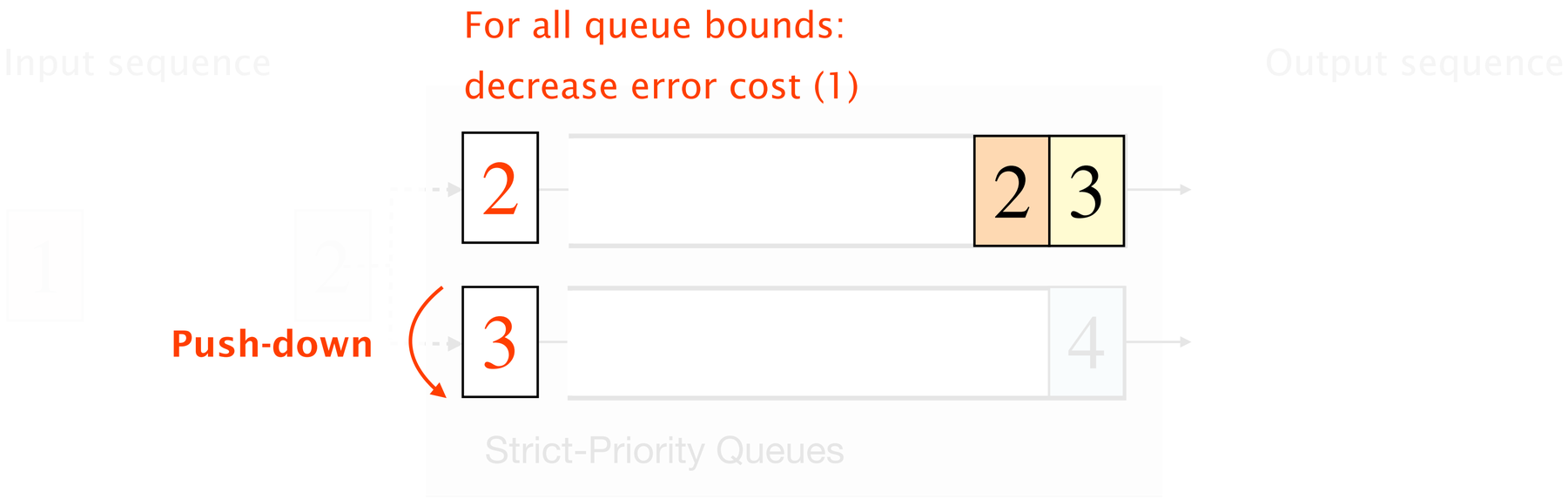
SP-PIFO adapts the mapping of packet ranks to strict-priority queues



SP-PIFO adapts the mapping of packet ranks to strict-priority queues



SP-PIFO adapts the mapping of packet ranks to strict-priority queues

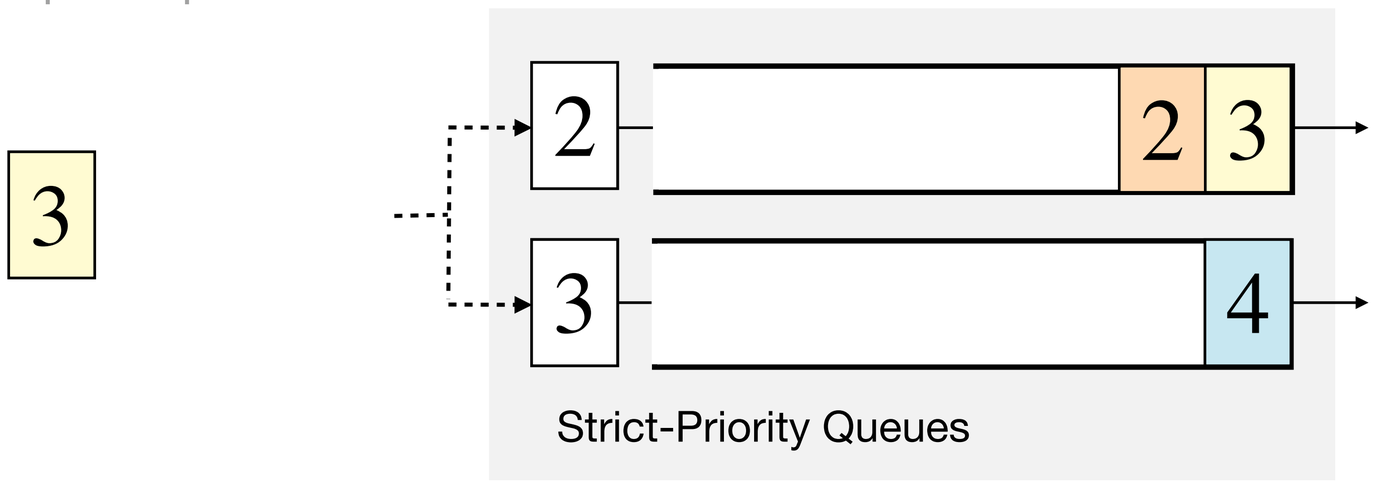


SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Push-down
(future high-rank packets to lower-priority queues)

After potential error detected,
all queue bounds decreased the error cost

Input sequence

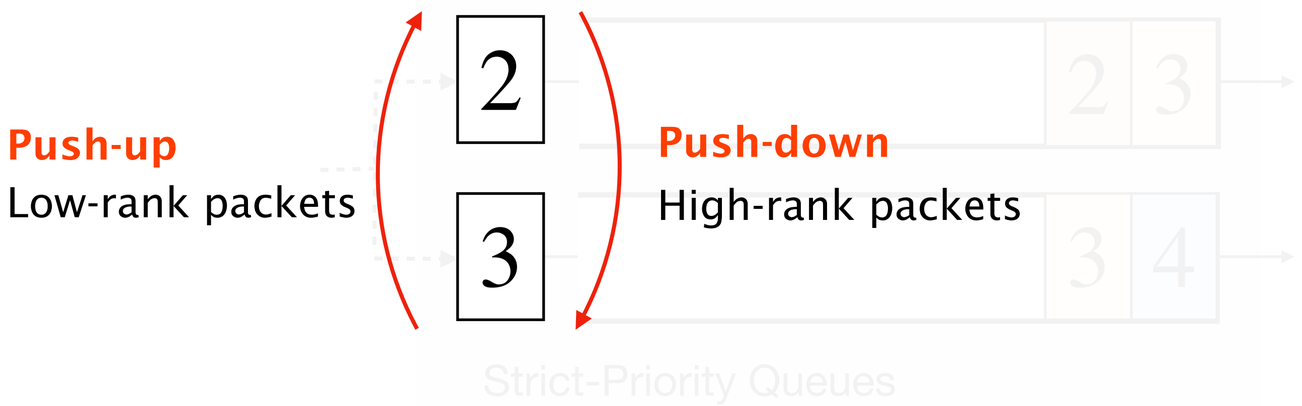


Output sequence

SP-PIFO adapts the mapping of packet ranks to strict-priority queues

Objective Find optimal queue bounds q^*
That minimize the expected loss U for all ranks

Result Packet-level adaptation of q



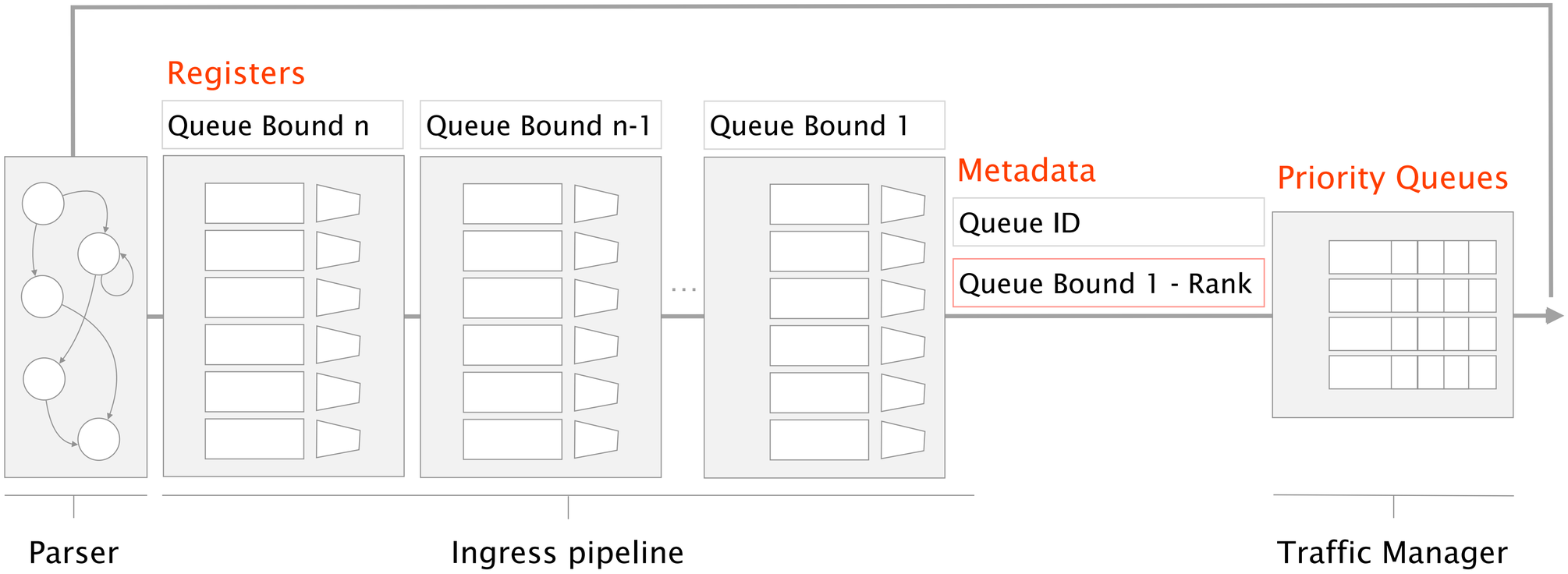
SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

1 Adaptation design
How does it work

2 Implementation
How can it be deployed

3 Evaluation
How well does it perform

SP-PIFO has been fully implemented on existing programmable hardware



SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

1 **Adaptation design**
How does it work

2 **Implementation**
How can it be deployed

3 **Evaluation**
How well does it perform

Evaluation

Question

How well does SP-PIFO approximate well-known scheduling objectives under realistic traffic workloads?

Scheduling objectives

Minimizing Flow Completion Time

pFabric* (8 queues)

Ranks are set to the remaining flow size

Max-min fairness

Start-Time Fair Queuing (32 queues)

Ranks based on a fluid model

* without starvation prevention

Methodology

Packet-level simulator

We integrated SP-PIFO in Netbench

[SIGCOMM 2017]

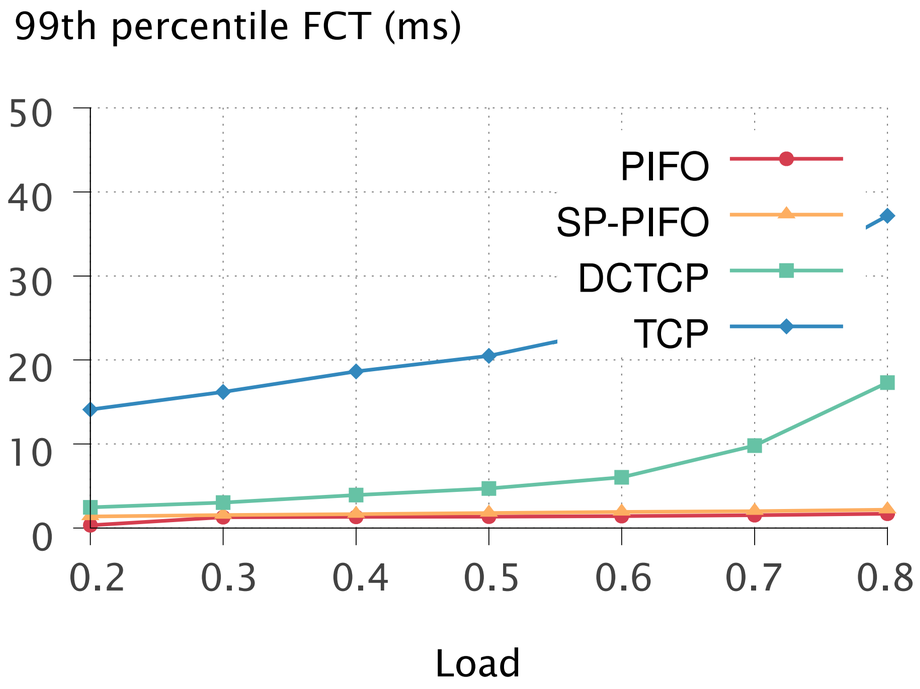
Topology

We use a leaf-spine topology with 144 servers, links of 1Gbps and 4Gbps

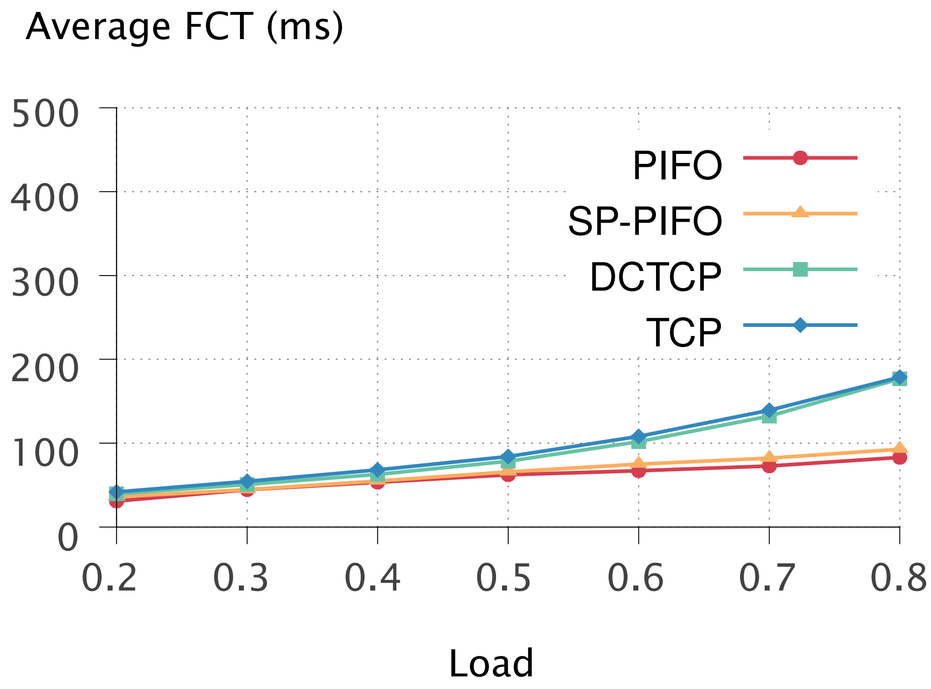
Realistic workloads

We generate traffic following pFabric web-search workload

SP-PIFO closely approximates pFabric, minimizing FCTs for both small and big flows

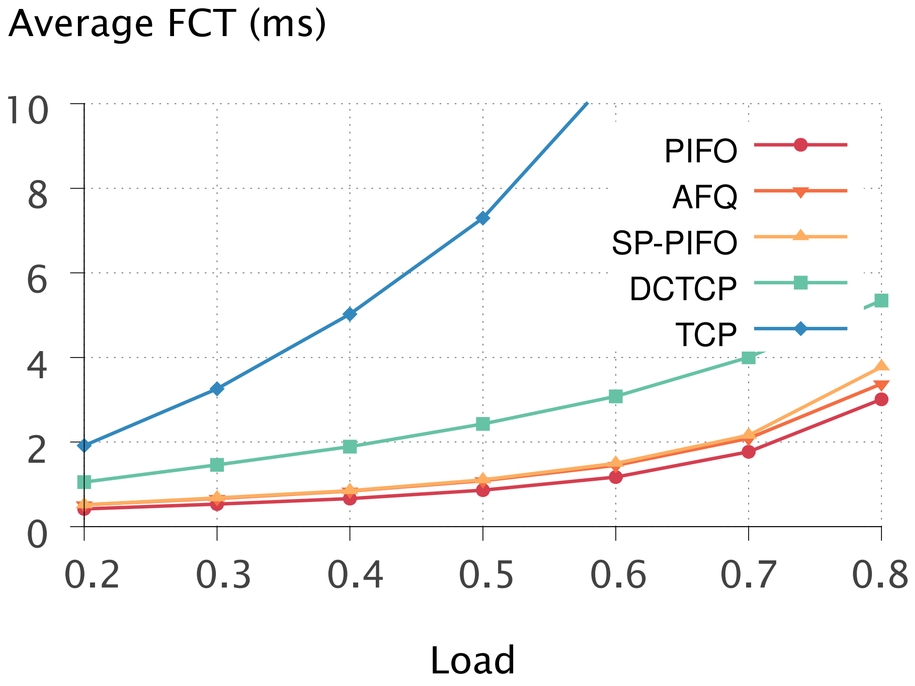


Small flows <100KB

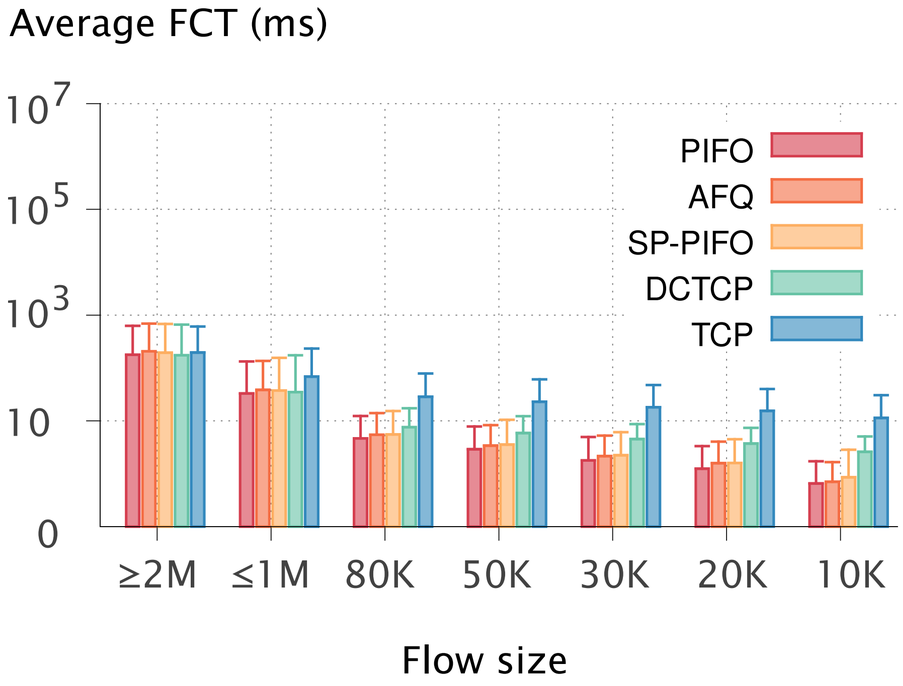


Big flows ≥1MB

SP-PIFO closely approximates state-of-the-art fair-queuing algorithms



Small flows <100KB



All flows @ Load 0.7

Check our website!

sp-pifo.ethz.ch

SP-PIFO characterization,
comparison with gradient

Hardware evaluation on
Barefoot Tofino

Limitations and
future improvements

All the code is available
All our experiments are reproducible

SP-PIFO: Making scheduling programmable, today!

SP-PIFO approximates the behavior of PIFO queues
at line rate, at scale and on existing devices

It adapts the mapping between packet ranks and
strict-priority queues to minimize the scheduling errors

It reacts per-packet to traffic variations,
without traffic knowledge required

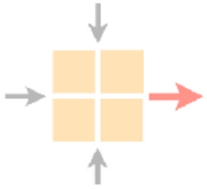
SP-PIFO: Approximating Push-In First-Out Behaviors Using Strict-Priority Queues

sp-pifo.ethz.ch

Albert Gran Alcoz

Alexander Dietmüller

Laurent Vanbever



Networked Systems
ETH Zürich — seit 2015

ETH zürich