Towards Software-Defined Buffer Management

Kirill Kogan, *Member, IEEE*, Danushka Menikkumbura, Gustavo Petri, Youngtae Noh, *Member, IEEE*, Sergey I. Nikolenko, Alexander Sirotkin, and Patrick Eugster

***Abstract*— Buffering architectures and policies for their efficient management are core ingredients of a network architecture. However, despite strong incentives to experiment with and deploy new policies, opportunities for changing anything beyond minor elements are limited. We introduce a new specification language, OpenQueue, that allows to express virtual buffering architectures and management policies representing a wide variety of economic models. OpenQueue allows users to specify entire buffering architectures and policies conveniently through several comparators and simple functions. We show examples of buffer management policies in OpenQueue and empirically demonstrate its impact on performance in various settings.**

***Index Terms*— Admission control, computer buffers, scheduling algorithms.**

I. Introduction

Buffering architectures define how input and output ports of a network element are connected [2], [3]. Their design and management directly impact performance and cost of each network element. Traditional network management only allows to deploy a predefined set of buffer management policies whose parameters can be adapted to specific network conditions. The incorporation of *new* management policies requires complex control/data plane code changes and sometimes respin of implementing hardware. Objectives beyond *fairness* [4]–[9] and additional traffic properties lead to new challenges in the implementation and performance for traditional switching architectures. Unfortunately, current developments in software-defined networking mostly eschew these challenges and concentrate on flexible and efficient representations of *packet classifiers* (e.g., OpenFlow [10], P4 [11]) that do not capture buffer management aspects well. This calls for new well-defined abstractions that enable buffer management policies to be deployed on real network elements at runtime, without changes in software or hardware.

Designing such abstractions is nontrivial, as they must satisfy possibly conflicting requirements: (1) EXPRESSIVITY: expressible policies should cover a large majority of existing and future deployment scenarios; (2) SIMPLICITY: policies for different objectives should be expressible concisely with a limited set of basic primitives and should not impose specific hardware choices; (3) PERFORMANCE: implementations of policies should be efficient on "virtual switches", with various resolutions ranging from a single network element to the whole network (e.g., an interconnect for geographically distributed data centers [7]–[9]); (4) DYNAMISM: new policies should be feasible to specify at run-time without any code changes and (re-)deployments. We address these tradeoffs by investigating in a *top-down manner* abstractions needed to systematically capture buffering management policies — including admission control, processing, and scheduling — and propose a crisply defined set of abstractions which account for efficient deployment by preserving line-rate characteristics.

Manuscript received August 21, 2019; revised March 10, 2020 and May 29, 2020; accepted July 13, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor P. Giaccone. Date of publication August 6, 2020; date of current version October 15, 2020. This work was supported in part by the Israeli Innovation Authority under the Knowledge Transfer Commercialization Program (MAGNETON) File No. 71249, in part by the Ariel Cyber Innovation Center in cooperation with the Israel National Cyber Directorate in the Prime Minister's Office, in part by the Data Science and Artificial Intelligence Research Center at Ariel University, in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education under Grant NRF-2020R1A4A1018774, in part by the Russian Science Foundation under Grant 17-11-01276, in part by the ERC Consolidator Grant #617805 (LiveSoft), in part by DFG Center #1053 (MAKI), and in part by SNSF Grant #200021 192121. This version is a significantly extended version of the article that appeared in IEEE ICNP'17. *(Corresponding author: Kirill Kogan.)*

Kirill Kogan is with the Department of Computer Science, Ariel University, Ariel 40700, Israel (e-mail: kirillk@ariel.ac.il).

Danushka Menikkumbura is with the Department of Computer Science, Purdue University, West Lafayette, IN 47907 USA (e-mail: dmenikku@purdue.edu).

Gustavo Petri is with Arm Research, Cambridge CB1 9NJ, U.K. (e-mail: gpetri@liafa.univ-paris-diderot.fr).

Youngtae Noh is with the Department of Computer Science and Information Engineering, Inha University, Incheon 402-751, South Korea (e-mail: ytnoh@inha.ac.kr).

Sergey I. Nikolenko is with the National Research University Higher School of Economics, 190008 St. Petersburg, Russia, and also with the Steklov Institute of Mathematics at St. Petersburg, 191023 St. Petersburg, Russia (e-mail: sergey@logic.pdmi.ras.ru).

Alexander Sirotkin is with the National Research University Higher School of Economics, 190008 St. Petersburg, Russia, and also with the St. Petersburg Institute for Informatics and Automation of the RAS, 199178 St. Petersburg, Russia (e-mail: alexander.sirotkin@gmail.com).

Patrick Eugster is with the Faculty of Informatics, Università della Svizzera Italiana (USI), 6900 Lugano, Switzerland, also with the Department of Computer Science, Purdue University, West Lafayette, IN 47907 USA, and also with the Department of Computer Science, TU Darmstadt, 64289 Darmstadt, Germany (e-mail: patrick.thomas.eugster@usi.ch).

Digital Object Identifier 10.1109/TNET.2020.3011048

II. Design Overview

To specify an adequate language for software-defined buffer management, we need to identify primitive entities, their properties, and a logic for manipulating them. The choice of primitives dictates the SIMPLICITY and EXPRESSIVITY of the language. For example, the main primitives in OpenFlow are *flows*, *actions*, etc., flow properties are *fields*, and the logic to manipulate flows is an ordered set of *conditions*. However, abstractions of ordered sets of conditions and actions governing the handling of individual incoming packets as in OpenFlow are insufficient for a language for buffer management. Packet classifier languages such as P4 are good in static

1063-6692 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

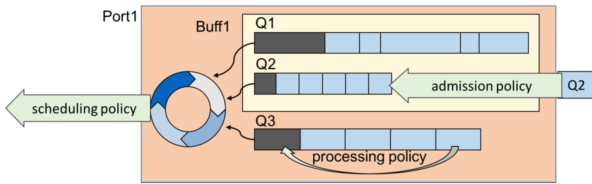


Fig. 1. Buffering architecture schema.

interrelations between patterns, but only for a single packet. Encoding inter-packet properties in classifiers built for single packet processing can be inefficient due to both a large number of static interrelations and high operational complexity of multi-field packet classifiers.

A. From Single Packets to Streams

Data plane services can be split into two major categories: (1) services that change properties of individual packets (e.g., recoloring fields, changing encapsulations, access control lists) and (2) services that change inter-packet properties (e.g., rate-limiting, shaping). The former, and some of the latter (e.g., rate-limiting), can be expressed efficiently (in terms of the amount of state required on the data plane) as hierarchical packet classifiers with action sets, i.e., "programs" that consist of an ordered set of conditions; the first condition matching an incoming packet defines the set of actions to apply to it. While classifiers also define a packet stream whose inter-packet properties can be changed, implementations of many type (2) services need to deal with *multiple packets together*: the current packet can affect already buffered packets. For example, in case of congestion one can drop the head-of-line (HOL) packet, e.g., with the Longest-Queue-Drop (LQD) policy in shared memory switches. Representing buffer management policies as a sequence of ordered conditions with actions on a currently "processed" packet is thus inefficient at best, and often infeasible, requiring as many conditions as the number of packets buffered. Since queueing modules can be implemented in specialized hardware, it is very important to find an expressive yet concise way to specify buffer management policies for different architectures.

B. Primitives

Figure 1 shows an abstract view of a network element's buffering architecture. OpenQueue has two main types of objects: *ports* and *queues* assigned to ports; in Fig. 1, `Port1` has three queues, `Q1`-`Q3`. Note that cross-points in the buffered crossbar architecture [12] can also be represented as ports. Each queue has an *admission control policy* that chooses packets to admit or drop [13]–[15]; in Fig. 1, an incoming packet marked for queue `Q2` has to be evaluated by `Q2`'s admission policy before being admitted. Each port defines a *scheduling policy* used to select a queue whose HOL packet will be processed next [16], [17]; in each queue, the HOL packet (darker in Fig. 1) is defined by a *processing policy*. In some cases, e.g., shared memory switches [18], several queues share the same *buffer space*, and the admission control

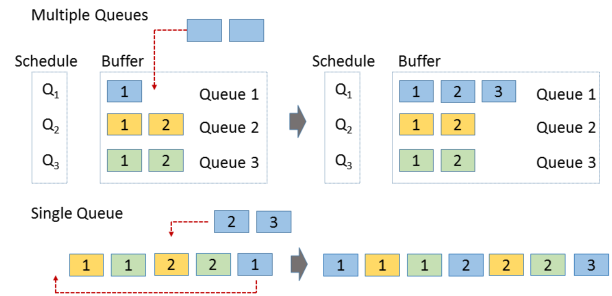


Fig. 2. Operational time complexity: the MQ architecture vs. SQ.

policy can routinely query the state of several queues; for instance, the LQD policy under congestion drops packets from the longest queue [18]. To capture these architectures, OpenQueue deals with buffers and admission control policy to resolve congestion at the buffer level; Fig. 1 shows a buffer `Buff1`. Management policies for multi-level buffering architectures can be implemented in a centralized or distributed manner, synchronously (e.g., finding a matching between input and output ports) [19], [20] or asynchronously, like packet scheduling in a buffered crossbar switch [12], [21], specific *implementations* are beyond the scope of OpenQueue. Thus, a buffering architecture and its management in OpenQueue is defined by instances of ports, queues, and buffers, and relations among them: admission control, processing, and scheduling policies.

C. Single or Multiple Queue Architectures

One of the central primitives in our language is the *queue*. But how complex should the queue abstraction and implementation be to achieve EXPRESSIVITY? In contrast to [22], which explores specifications of universal scheduling policies that can satisfy multiple objectives, we argue for separating admission, processing, and scheduling policies, and supporting multiple (as well as single) queues, through some novel fundamental results. Consider first a single queue (SQ) buffering architecture of size B, uniformly sized packets with individual values, and the objective of maximizing the total transmitted value (weighted throughput). Usually, online buffer management policies are evaluated by means of competitive analysis [2], [23], where an online policy is compared with an offline optimal algorithm. Traditionally, queues implement First-In-First-Out (FIFO) processing orders; can we find an optimal online algorithm in this case?

Theorem 1: There is no deterministic online optimal algorithm for the SQ architecture, weighted throughput objective, and FIFO processing.

Proof: Assume by contradiction that such optimal algorithm OPT exists. Consider an initial arrival of two packets with values of 1 and 2, in that order. In the first timeslot, OPT has to either process or discard the first packet with value 1. If it discards it, there are no further arrivals, and OPT transmits less than an algorithm that accepts both packets. If OPT accepts a packet with value 1, on the second timeslot there arrive B packets of value 2 each, and OPT loses to an

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

algorithm that discarded the first packet and is now able to accept all new arrivals. □

Consider another online policy PQ_1 for SQ that processes the most valuable packet first and in case of congestion pushes out least valuable packets.

Theorem 2: PQ_1 is better than any online deterministic policy $FIFO_1$ with FIFO processing.

Proof: First, note that for packets with the same processing time of one timeslot, PQ_1 drops no more packets than any other online algorithm (all greedy algorithms are congested at the same time). Since PQ_1 transmits the most valuable packets and drops the least valuable, PQ_1 is no worse than $FIFO_1$. To show that there are inputs where PQ_1 is strictly better than $FIFO_1$, consider an arrival sequence with two packets with values 1 and 2, in that order. PQ_1 transmits a packet of value 2. If $FIFO_1$ discards the unit-valued packet, it will transmit less than PQ_1 if there are no future arrivals. If $FIFO_1$ accepts and transmits the unit-valued packet, on the next timeslot B packets of value 2. In this case PQ_1 transmits total value $2(B+1)$ but $FIFO_1$ transmits only $2B+1$. □

PQ_1 is the best possible policy for packets with values and weighted throughput, regardless of arrival patterns. But its processing order may be infeasible for some existing network elements designed with FIFO in mind. This example motivates the ability to abstract/modify processing order, as well as admission control policy to define push-outs. Consider next a multiple queue (MQ) architecture with the same buffer size as SQ where each queue has FIFO processing and is dedicated to packets with the same value (possibly the same value for several queues). This simplifies processing and moves complexity to scheduling decisions. The algorithm that pushes out the least valuable and schedules the most valuable packet transmits the same total value as PQ_1.

Corollary 1: There exists an online deterministic algorithm for MQ with FIFO at every queue that is better than any deterministic online $FIFO_1$.

This motivates multiple queues, which in turn demands for a scheduling policy in order to choose between different queues. Our design choices are independent of specific objectives and packet characteristics. Consider again the MQ buffering architecture with m queues with any processing order allowed and admission control at every queue. As before, to better handle bursty traffic we assume that the queues share a buffer of size B [24]. In this case the MQ architecture has an additional level of flexibility during scheduling (in addition to admission control and processing). It turns out that for the same buffer size B they are equally expressive.

Theorem 3: For any deterministic (or probabilistic) MQ policy ALG, there exists an SQ policy that for any input sequence transmits exactly the same set of packets (or a set of packets with same distribution) as ALG.

Proof: The decisions of ALG depend only on the internal buffer state and random bits. Therefore, this information suffices to recreate the processing order in SQ if there are no new arrivals. This order can be used as a priority function in the SQ architecture. Now the SQ policy emulates ALG's behaviour exactly and chooses the packet to process according to MQ's decisions, recalculating priorities on every arrival. In the stochastic case, policies may diverge in each specific case due to randomness, but as long as the probabilities of SQ decisions are the same as ALG's for the same buffer state, the resulting distributions coincide. □

Theorem 3 shows that in theory, SQ is as expressive as MQ for any objective and any combination of packet characteristics for both deterministic and stochastic policies. So do we really need multiple queues on a single output port (assuming no physical constraints such as memory access bandwidth)? In fact, the work [25] recently proposed to express policies with a single priority queue and a single calendar queue. However,

additional constraints and parameters such as the time complexity of operations may arise that can make one buffering architecture preferable.

Theorem 4: In the worst case, the MQ architecture with m queues has time complexity of operations at least $O(m/\log B)$ times better than the SQ architecture simulating the same order.

Proof: To show the worst-case bound, we need a specific example where multiple queues are hard to simulate efficiently. Suppose that every arriving packet has an associated queue number that corresponds to a flow identifier (but no other characteristics), each queue has to preserve FIFO ordering, and the policy objective is to ensure fairness between queues, i.e., the multiple queue policy is Longest-Queue-First (LQF). Note that arrivals for MQ cost $O(\log m)$ to choose the queue and $O(\log B)$ to add to the queue implemented as a priority queue; updates in a FIFO queue with identical packets might cost $O(1)$ but here we have sacrificed some efficiency for the generality of our scheme. Now by adding one or two packets we can completely reorder the single queue that emulates LQF (see Fig. 2). For a specific example, consider m queues and $B > m^2$; at the first burst m packets arrive for every queue, say $p_{i,1}, \ldots, p_{i,m}$ to queue i. SQ now has interleaved packets: $p_{1,1}p_{2,1}\ldots p_{m,1}p_{1,2}\ldots p_{m,2}p_{3,1}\ldots p_{m,m}$. After the first timeslot, $p_{1,1}$ has been processed and left the buffer, and two more packets arrive in queue 1, $p_{1,m+1}$ and $p_{1,m+2}$. Now all packets from queue 1 have to be reordered to move forward in the single queue; the SQ order now has to become $p_{1,2}p_{2,1}\ldots p_{m,1}p_{1,3}p_{2,2}\ldots p_{m,2}p_{3,1}\ldots p_{m,m}p_{1,m+2}$, which takes at least m operations even if we assume $O(1)$ operations in SQ. After the next timeslot, all queues again have the same number of packets, and the sequence can be repeated; note that the buffers are never congested in this example. □

Based on the above observations, OpenQueue allows attaching multiple queues to the same port that share or do not share the same buffer; naturally, this allows for single queues as a special case. But how to adequately express policies? Buffer management policies are generally concerned with *boundary conditions* (e.g., upon admission a packet with *smallest* value can be dropped). Hence, *priority queues* arise as a natural choice for implementing actions related to user-defined priorities (e.g., in FIFO processing order, a packet with *smallest* arrival time is chosen next). The priority criteria do not change at runtime (e.g., a queue's ordering cannot change from FIFO to LIFO). Thus, each admission, processing, and scheduling policy in OpenQueue maintains its priority queue data structure whose behavior is defined by a simple

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

comparator – a Boolean function comparing two objects of same type via arithmetic/Boolean operators and accesses to packet and object attributes. In addition, to specify when a queue or buffer should be considered congested, we introduce simple Boolean *conditions*. These simple constructs reconcile EXPRESSIVITY and SIMPLICITY, retaining PERFORMANCE, e.g., guaranteeing a constant number of insert/remove and lookup operations during admission or scheduling of a packet.

III. OpenQueue Specification Language

Below, we present in detail the interface OpenQueue provides to a programmer for specifying a buffering architecture and its management policy, aiming to reconcile SIMPLICITY and EXPRESSIVITY while keeping PERFORMANCE in mind (cf. Section I). The fundamental primitives to be considered are *packet*, *queue*, *port*, and *buffer* (see List 1). *Packets* represent an abstract view of the actual packet properties; in addition, packets in OpenQueue can carry data set by external packet classifiers implemented in a language like P4 [11] or OpenFlow [10]. *Queues* define which packets are admitted and their processing order. *Ports* define how to allocate bandwidth among different queues. *Buffers* are optional entities, defined only if several queues share the same buffer space as in shared-memory switches [18]. For each primitive, we provide its properties. Some are properties of the domain (e.g., packet size), and others have to be set by the programmer. Unassigned properties get default values to guarantee consistent behavior. For each property, we indicate whether it is read-only (**`r`**) or also writable (**`rw`**), whether its value is fixed during execution (**`cons`**), or not (**`dyn`**). The logic of a buffer management policy is based on *comparators* and *boundary conditions*.

A. Packets

All packet properties are set by an external classification module except two: *arrival* time and *size* are set on arrival by the queueing module. List. 1 depicts the **`Packet`** primitive. As packets cannot be affected by an OpenQueue policy programmer, there is no constructor for them. The `policyId` field is used during policy changes under traffic to guarantee consistency. The `queue` allows to route a packet inside the buffering architecture. Intrinsic `value` (with application-specific meaning) and `processing` requirements (in virtual cycles) are used to define prioritization levels [26], [27]. The `slack` is a time bound used in management decisions of latency-sensitive applications; e.g., if buffer occupancy already exceeds the slack value of an incoming packet, the packet can be dropped during admission even if there is available buffer space [28]. Field `flow` allows to distinguish among packet streams at a resolution higher than the queue level. See Section IV for specific examples. We posit that all decisions of buffer management policies (admission, processing, or scheduling) are based only on specified packet parameters and internal state variables of a buffering architecture.

B. Comparators

The *priority queue* is the core data structure for the expression of buffering architectures. To express queues with *different priorities*, while abstracting actual implementations, many data structures in OpenQueue are parameterized by a priority relation that determines the ordering of elements in a queue. To that end, we introduce the notion of a *comparator*, a Boolean binary predicate (parameterized on the types of the arguments). Since comparators are used internally by OpenQueue to implement queues, the comparison operation has to be efficiently computable (ideally at the hardware level). To achieve efficient computation with comparators, OpenQueue thus imposes certain syntactic restrictions on their definition. The syntax below captures the main restrictions.

x		formal variables
n		numeric constants
e	$::= n \mid x.f \mid e \oplus e$	arithmetic expressions
b	$::= e \otimes e \mid b \oslash b$	predicates
c	$::= comp_name(x,x) = b$	comparator def.

Comparator declarations take a name $comp_name$, and two more arguments. The type of the arguments varies depending on the priority being defined. For queues which hold packets, for instance, a packet comparator has to be defined. The fourth production of the grammar contains predicates, which are the Boolean expressions b defining the comparison function. Aside from the standard arithmetic expressions (we generically denote by $\oplus$ the standard arithmetic operators), the first-order Boolean operators (denoted with the symbol $\oslash$) and arithmetic relations (denoted with $\otimes$), we allow the inspection of the fields of the arguments using the standard dot notation $x.f$, where f is assumed to be a field of parameter x. It is assumed here that field access operations require a small constant number of memory accesses (generally one). Importantly, no function or procedure calls are allowed in comparators. E.g., to implement first-in-first-out packet order, it is sufficient to order packets by arrival time:

```
fifo(p1, p2) = (p1.arrival < p2.arrival)
```

We assume that a priority queue data structure supports, in addition to `insert()` for adding a new element, `extractHigh()` and `findHigh()` methods for extracting and finding an element with highest priority, respectively.

C. Boundary Conditions

Data structures manipulated by OpenQueue can behave differently. For instance, the user could specify that certain packets should be dropped to achieve graceful degradation when a queue becomes saturated. Again, we use a restricted language to express these *boundary conditions*. Unlike comparators, the predicates of boundary conditions are *unary* since they consider a single entity at any time.

pf	$::=$ `admState` | `schedState`	modifiables
ac	$::=$ **`drop`**(P) | **`modify`**$(pf := e)$	actions
	| **`mark`** | **`notify`** | $ac \cdot ac$	
cl	$::= (b, ac) \mid (b, ac) \cdot cl$	condition cases
cd	$::= cond_name(x) = cl$	declarations

This syntax shares the definitions of predicates and declarations with comparators seen before. Importantly, a boundary condition can be a sequence of cases (each case separated with a dot above). This is represented by the cl meta-variable representing a list of pairs, whose first component contains a

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

predicate and second component defines an *action* represented by the meta-variable ac.[1] The **drop**(P) action indicates that packets have to be dropped from the queue with probability P if the matching predicate evaluates to true. For example, the following implements tail drop in a queue q, where `currSize` and `size` are the current and maximal queue sizes, respectively (queue properties are defined in Section III-D).

```
tailDrop(q) = q.currSize > q.size, drop(1)
```

The **modify**($pf := e$) action changes writable packet properties and internal state of a managed buffering architecture. The **notify** action is used to implement backpressure by setting *explicit congestion notifications* [29]. **mark** allows receivers to differentiate traffic similarly to *cell loss priority* or *discard eligibility*. Moreover, we allow sequences of actions, although generally only one action is used in conditions. Condition cases enable the expression of different response scenarios. For example, under severe congestion a more aggressive drop policy can be put in place by increasing the probability of dropping a packet. It is best if the conditions are mutually exclusive; in the current version of OpenQueue only actions of the first matching condition (in lexicographic order) will be triggered.

In the following, we introduce three building blocks of OpenQueue: **Queue**, **Port** and optional **Buffer** allowing to define buffering architectures and their managing policies whose logic is based on a fixed number of comparators and boundary conditions. For inline functions, we provide the return type (e.g., **bool fun**), and we denote comparators indicating their input types (e.g., **Packet comp.**), and boundary conditions indicating the actions that they allow (e.g., **drop cond**).

```
Packet {
// set by external packet classifier
policyId //used on policy changes [r, cons]
queue // target queue id [r, cons]
value // virtual value [rw, dyn]
processing // no of cycles [r, dyn]
slack // offset in time [r, cons]
flow // flow id [r, cons]
// set on arrival
arrival // arrival time [r, cons]
size // size in bytes [r, cons]
}

Queue {
Queue(size) // constructor
size // declared size in bytes [r, cons]
currSize // current queue size [r, dyn]
// admission policy-- user-specified at decl.
admPrio(p1, p2) // pushOut comparat.[bool fun]
congestion()//[{drop,mark,notify,modify} cond]
admState // admission state [rw, dyn]
// processing policy -- user-spec. at decl.
procPrio(p1, p2)// proc comparat.[Packet comp]
schedState // per-queue sched. state [rw, dyn]
}

Port {
Port(rate, q1, .., qk) // constructor
rate // declared rate in
// scheduling policy, user-specified at decl.
schedPrio(q1, q2)// compare q-s [bool fun]
schedState() // scheduling state [rw, dyn]
postSchedAct() //[{mark,notify,modify} cond]
currQueue //current queue [rw, dyn]
}

Buffer {
Buffer(size, q1, .., qk) // constructor
size // declared size in bytes [r, cons]
currSize // current queue size [r, dyn]
currQueue // current queue [rw,dyn]
// admission policy-- user-specified at decl.
admPrio(q1, q2) // pushOut comparat.[bool fun]
congestion()//[{drop,mark,notify,modify} cond]
admState // optional per-buffer state [rw, dyn]
}
```

Listing 1. OpenQueue's core programming interface at a glance: packet, queue, port, and buffer primitives.

D. Queues

List. 1 shows the declaration of queues.

The standard property `size` is defined by the user at declaration time.

```
q = Queue(B) //creates a queue of size B
```

A packet enters a queue identified by the packet queue field. The `currSize` property serves to query the queue size and changes dynamically as the queue is updated. Queue internals consist of `admission` and `processing` policies, each maintaining a priority queue data structure.

Admission policy defines which packets are preferred during "congestion", specified in `congestion()` boundary condition and `admState`. The single argument of boundary condition declarations is implicitly instantiated to the queue being defined, hence, this is a queue boundary condition. The deterministic drop action here corresponds to the **drop**(1) action in the syntax presented in Section III-C. Usually, `congestion()` is a list of queue occupancies and corresponding drop probabilities [13]. The example below shows a possible "RED-like" congestion condition where packets start being dropped with probability .5 if the current queue occupancy is greater than 3/4 of the total queue size but lower than 9/10; they are dropped with probability .9 for occupancies between 9/10 and 19/20 [13]. Otherwise, packets are always dropped and backpressure is signalled later.

```
red(q) =
(q.currSize >= .95*q.size, drop(1), notify).
(q.currSize >= .9*q.size, drop(.9)).
(q.currSize >= .75*q.size, drop(.5))
```

The packet comparator `admPrio(p1, p2)` defines which packets are to be dropped from the queue during congestion. For example, the user can set

```
admPrio(p1, p2) = (p1.value < p2.value)
```

to keep the most valuable packets in case of congestion. OpenQueue supports the capability to *push out* already admitted packets. To use the same implementation for push-out and non-push-out cases, an admission control policy always virtually admits incoming packets. In the event of virtual congestion, admission control probabilistically drops the least valuable packets until the congestion condition is lifted. The following congestion boundary condition shows a simple use-case of `admState` for dropping every third incoming packet:

```
third_drop(q) = (mod(q.admState, 3) == 0,
modify(q.admState++), drop(1)).
(true, modify(q.admState++))
```

[1]The syntax presented here is simplified for presentation purposes.

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

Processing policy defines a packet processing order inside a queue according to `procPrio(p1, p2)`. As an example, the user can set

```
procPrio(p1, p2) = (p1.arrival < p2.arrival)
```

to encode FIFO processing.

In addition, **`Queue`** maintains per-queue state `schedState` for the *scheduling policy* defined on the port level (see Section III-E).

E. Ports

The interface for ports is presented in List. 1. The standard property `rate` is defined by a user at declaration time as a percentage of the corresponding physical rate or in kbps. A port manages a set of queues assigned to it at declaration. Below, two queues are attached to a port:

```
q1 = Queue(B); q2 = Queue(B);
out = Port(100
```

Scheduling policy defines which queue will transmit according to `schedPrio(q1, q2)` priority. E.g., priority based on packet values with several levels of strict priorities is defined as:

```
schedPrio(q1, q2) =
(q1.findHigh().value > q2.findHigh().value)
```

Finally, `postSchedAct()` is a boundary condition similar to `congestion()` at the queue level allowing to run actions at port resolution rather than at the queue level. For instance, when bandwidth among queues is allocated not only with respect to packet attributes, queues maintain a `schedState` variable that can be updated dynamically after each packet scheduling with **`modify`**`()` action.

F. Buffers (Optional)

A buffer is an optional entity, declared only when several queues share buffer space and an additional level of the admission control on this buffer is necessary (see List. 1). The standard property `size` is defined by the user at declaration time. Similarly to **`Port`**, **`Buffer`** manages a set of queues assigned to it at creation. For instance, the following lines in OpenQueue define a shared-memory switch with three output ports, one queue of size `B` attached to each port, and all queues share a buffer of size `B` (each queue can occupy the whole buffer space).

```
q1 = Queue(B); q2 = Queue(B); q3 = Queue(B);
p1 = Port(100
p3 = Port(100
```

Admission policy at the buffer level in the case of congestion defined by `congestion()` finds a queue whose packet should be dropped according to `admPrio(q1, q2)` priority and `admState`; and the actual packet is dropped according to the chosen queue `admPrio(p1, p2)`. Note that in this case the congestion condition on the buffer level has precedence over that of the chosen queue. For example, to implement Longest-Queue-Drop(LQD) [18] on the shared-memory switch defined above, the following suffices:

```
longest(q1, q2) = (q1.currSize > q2.currSize);
bufDrop(b) = b.currSize > b.size, drop(1);
b.admPrio = longest();
b.congestion = bufDrop();
```

By default `procPrio` is `fifo()`, `admPrio` is reverse FIFO, and `congestion` is `tailDrop()`.

G. Composing and Verifying

We assume that a *verified* implementation of a priority queue data structure with efficient `insert()`, `findHigh()`, and `extractHigh()` operations is available. Moreover, we assume that the OpenQueue compiler itself is verified. To avoid errors related to unassigned properties, we use default values for all properties. That is, buffering policies are expressed by instantiating data structures via constructors, shown in List. 1, and connecting these, or by explicitly assigning values to properties of data structures. There are two mandatory fields to set at every packet p by external classifiers: **`queue`** specifying to which queue in the specified buffering architecture p should be added and `policyId` referencing the current consistent buffer management policy. All unset packet fields and attributes of queues, ports, and buffers get default values if unset at construction. All policy changes are done in batches. The `policyId` value points to the consistent version of a policy. Default values for unset attributes, batch changes and `policyId`, guarantee correctness of policy changes under traffic. A typical chain of instantiation is to set up queues, then buffers and ports, which the latter two stages being parameterized by data structures from the first stage. Verification of constraints such as post admission or scheduler actions (cf. **`Buffer`**, **`Port`** respectively) only using **`mark`**, **`notify`**, and **`modify`** actions happens quite easily at such implicit (constructor) or explicit assignments, as all declarations have to be present, and we avoid aliasing. Corresponding extensions and relaxations, e.g., for allowing specifications to be compiled piece-wise, are the subject of ongoing work.

H. Expressiveness vs. Operational Complexity

A fundamental building block in OpenQueue is the *priority queue* (PQ) data structure based on a user-defined priority. Our implementation keeps a single copy of each packet and uses pointers to encode priorities (Fig. 5). In the most expressive case, we propose two PQs at the queue level (for admission and processing), one for scheduling at the port level, and another one in case where there is shared memory. The operational complexity of OpenQueue on a given target platform boils down to the efficiency of the underlying PQ implementation and the number of instances of PQs involved in a specified management policy. While in general PQ operations `insert`, `findHigh`, and `extractHigh` have $O(\log N)$ time complexity (N is the queue size), there are restricted versions (e.g., for predefined ranges of priority values) that support most operations in $O(1)$ and can be efficiently implemented even in hardware [25], [30], [31]. Another possibility is to have a more complex architecture while reducing per-packet operational complexity. In Section II-C we show that while a single queue (SQ) buffering architecture can be as expressive

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

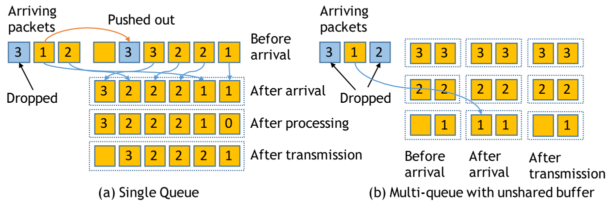


Fig. 3. Packets with required processing. Left: single priority queue with buffer of size $B = 6$; right: multiple separated queues with three queues ($k = 3$) of size 2 each. Dashed lines enclose queues.

as one with multiple queues at a single port (MQ), the MQ architecture can be significantly better in terms of operational complexity. Generally, for queues it makes sense to keep reversed priorities for processing and admission; for instance, to optimize weighted throughput preferring maximum packet values first but on congestion dropping minimal values. In this case, we can consider an efficiently implemented PQ data structure supporting two additional operations: `findLow` and `extractLow`. Moreover, even in the single queue case, fixing one of the priorities and implementing it natively on specific hardware can be an additional option; for instance, restricting processing to FIFO and implementing it natively, hence side-stepping the PQ implementation as we do in Section IV-C. We evaluate the majority of these tradeoffs in Section IV-G. To guarantee a constant number of insert/remove and lookup operations during admission or scheduling of a packet (i.e., to avoid rebuilding the priority queue), OpenQueue's user-defined expressions for priorities are immutable. To avoid performance degradation, policies requiring linear searches over enqueued packets, control message exchange during operations cannot be specified in OpenQueue; for instance, for (Combined Input- and Output-Queued (CIOQ) switches scheduling policies built on "request-grant-accept" principle (as PIM, iSLIP, etc. [32], [33]). The complexity of OpenQueue is hence reduced to translating user-defined settings to a target system that implements a virtual buffering architecture.

IV. Putting OpenQueue to Work

In this section we provide various examples of buffer management policies demonstrating EXPRESSIVITY, SIMPLICITY, as well as DYNAMISM of OpenQueue. In particular, we demonstrate the impact of each one of the admission control, processing, and scheduling policies on the desired objective. In addition we provide backward references to analytic results thus reconciling results from theory and systems efforts.

A. Impact of Admission Control

The modern network edge is required to perform tasks with heterogeneous complexity, including deep packet inspection, firewalling, and intrusion detection. Hence, the way packets are processed may significantly affect desired objectives. For example, increasing per-packet processing time for some flows can trigger congestion even for traffic with relatively modest burstiness.

```
// priorities for admission and processing
fifo(p1, p2) = (p1.arrival < p2.arrival)
rfifo(p1, p2) = (p1.arrival > p2.arrival)
srpt(p1, p2) = (p1.processing < p2.processing)
rsrpt(p1, p2) = (p1.processing > p2.processing)
// congestion conds. considered.
// trigger when occupancy exceeds size.
tailDrop(q) = q.currSize >= q.size, drop(1)
```

Listing 2. Example priorities and congestion conditions.

```
// buffering architecture specification
q1 = Queue(B);
out = Port(100
// admission control
q1.admPrio(p1, p2) = rsrpt(p1, p2);
q1.congestion = tailDrop(q1);
// processing policy
q1.procPrio(p1, p2) = srpt(p1, p2);
```

Listing 3. Single queue: optimal buffer management policy for throughput optimization.

TABLE I
SAMPLE OPENQUEUE POLICIES WITH ANALYTIC RESULTS FOR THE SQ ARCHITECTURE (k – MAX PROCESSING)

admPrio	procPrio	OPT/ALG
fifo()	fifo()	$O(k)$
rsrpt()	fifo()	$O(\log k)$
rsrpt()	srpt()	1 (optimal)

Consider throughput maximization in a single queue buffering architecture of size B, where each unit-sized and unit-valued packet is assigned the number of required processing cycles, ranging from 1 to k (see Fig. 3(a)). Defining a new admission control policy in OpenQueue requires only one comparator (admission order upon congestion) and one congestion condition (when an event of congestion occurs). The processing policy is defined by one additional comparator (defining in which order packets are processed). Note that admission and processing comparators actually can be different. List. 2 shows the comparators and congestion conditions specified in OpenQueue used in the following examples.[2] List. 3 shows the full specification of a SQ buffering architecture and its optimal throughput policy.

Table I lists implementations for `admPrio` and `procPrio` in this architecture and analytic competitiveness results for various online policies versus the optimal offline OPT algorithm [3], [26]. Each row represents a buffer management policy for a single queue; e.g., the first row shows a simple greedy algorithm that admits every incoming packet if possible (see `congestion()`), and processes them in `fifo()` order; it is $O(k)$-competitive for maximum processing requirement k. In OpenQueue this becomes simply:

```
q1.admPrio = rfifo();
q1.procPrio = fifo();
```

Changing `fifo()` admission order to `rsrpt()` significantly improves performance and this version of the greedy policy

[2]We use **let** constructs to make implicit variables explicit, and to avoid repetition. These are simple syntactic forms that affect in no way the EXPRESSIVITY or SIMPLICITY of OpenQueue.

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

```
// create k queues each of size B
q1 = Queue(B);..; qk = Queue(B);
out = Port(100
// fifo admission order
q1.admPrio = rfifo();..; qk.admPrio = rfifo();
// fifo processing order
q1.procPrio = fifo();..; qk.procPrio = fifo();
// congestion condition
q1.congestion = tailDrop(q1);..;
qk.congestion = tailDrop(qk);
```

Listing 4. Multiple separated FIFO queues with a single output port architecture.

TABLE II

SAMPLE OPENQUEUE POLICIES FOR MULTIPLE SEPARATED QUEUES (k – MAX PROCESSING, B – SIZE OF EACH QUEUE)

init. schedState	postSchedAct	schedPrio	OPT/ALG
unused	unused	lqf()	$\Omega(\frac{B}{2})$
unused	unused	sqf()	$\Omega(k)$
unused	unused	maxqf()	$\Omega(k)$
qi.schedState=i	unused	minqf()	≤ 2
qi.schedState=i	crrPostSchedAct()	crr()	$\Omega(\frac{k}{\ln k})$
qi.schedState=i	prrPostSchedAct()	prr()	$\Omega(\frac{3k(k+2)}{4k+16})$

is already $O(\log(k))$-competitive. With the third greedy algorithm processing packets in `srpt()` order and admitting them in `rsrpt()` order, we get an optimal algorithm for throughput maximization regardless of traffic distribution [26]. Since here a port manages only one queue, a *scheduling policy* is just an implicit call to `extractHigh()`.

B. Impact of Scheduling

One alternative architecture for packets with heterogeneous processing requirements is to allocate queues for packets with the same processing requirements (see Fig. 3(b)). The OpenQueue code in List. 4 creates this buffering architecture, with `k` separate queues of size `B`.

In this architecture, advanced processing and admission orders are not required as only packets with same processing requirements are admitted to the same queue. This change of buffering architecture is not for free because the buffer of these queues is not shareable. But even here, the decision of which packet to process in order to maximize throughput is non-trivial since it is unclear which characteristic (i.e., buffer occupancy, required processing, or a combination) is most relevant for throughput optimization. The code in List. 5 presents six different scheduling priorities and `postSchedAct` actions when these actions are used.

Table II summarizes various online scheduling policies as shown in [3], [34]. Observe that buffer occupancy is not a good characteristic for throughput maximization: `lqf()` and `sqf()` have bad competitive ratios, while a simple greedy scheduling policy Min-Queue-First (MQF) that processes the HOL packet from the non-empty queue with minimal required processing (`minqf()`) is 2-competitive. This means that MQF will have optimal throughput with a moderate speedup of 2 [34]. The other two policies that implement fairness with per-cycle or per-packet resolution (CRR and PRR respectively) perform relatively poorly; this demonstrates the fundamental tradeoff between fairness and throughput. The following code

```
// LQF: HOL packet from Longest-Queue-First
lqf(q1,q2) = (q1.currSize > q2.currSize);
// SQF: HOL packet from Shortest-Queue-First
sqf(q1,q2) = (q1.currSize < q2.currSize);
// MAXQF: HOL packet from queue that
// admits max processing
maxqf(q1,q2) = (q1.schedState > q2.schedState);
// MINQF: HOL packet from queue that admits
// min processing
minqf(q1,q2) = (q1.schedState < q2.schedState);
// CRR: Round-Robin with per cycle resolution
crr(q1,q2) = (q1.schedState < q2.schedState);
crrPostSchedAct(port) =
let q = port.getCurrQueue() in
(true, // condition
modify(q.schedState := q.schedState+k));
// PRR: Round-Robin with per packet resolution
prr(q1,q2) = (q1.schedState < q2.schedState);
prrPostSchedAct(port) =
let q = port.getCurrQueue() in
(q.findHigh().processing == 0, // condition
modify(schedState := schedState+k*k));
```

Listing 5. OpenQueue example of scheduling priorities and `postSchedAct` actions for multiple separated queues.

```
// initializing schedWeight for CRR
q1.schedState = 1; ...; qk.schedState = k;
// postSchedAct updating schedWeight
out.postSchedAct = crrPostSchedAct(out);
```

Listing 6. CRR policy for multiple separated queues.

snippet in OpenQueue, for instance, corresponds to the CRR policy:

C. Software-Defined Transports

Recently, new transports have been introduced to optimize various objectives [7], [35]. Some of them require complex processing orders and support of push-out, which can require complex code changes both on control and data planes. For instance, pFabric [7] prioritizes packets according to remaining flow completion time (FCT) and during congestion pushes out least valuable packets. In the original implementation of *pFabric*, on dequeuing a packet a linear search is done over the entire queue to find the highest priority packet from the flow in order to avoid reordering packets of the same flow. Although the *pFabric* paper mentions that evaluations do not encounter large queue sizes, on general workloads this can become a bottleneck. Since *pFabric* was evaluated in the YAPS discrete simulator [36], this linear search has no operational effect. In reality the situation can be different, and queue occupancy can significantly increase due to this overhead. To avoid this, we can push out the least valuable packet during congestion but process packets in FIFO order. Listing 7 shows this new transport in OpenQueue. Clearly, *pFabric* with FIFO cannot be better than the original *pFabric* in a simulation environment. But even in this environment the normalized FCT[3] is close, e.g., on the IMC10 workload (see Fig. 4). Note that our goal here is not to introduce another transport and describe its properties but to show the simplicity of OpenQueue and its ability to introduce new transports at run-time without any

[3]Normalized FCT is the ratio of the average FCT(i) and OPT(i), which is the FCT if this was the only flow in the network [35], [37].

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

```
// buffering architecture specification
q1 = Queue(B);
out = Port(100
// admission control: by remaining flow size
q1.admPrio(p1, p2) = (p1.value > p2.value);
q1.congestion = tailDrop(q1);
// processing policy: fifo()
q1.procPrio(p1, p2) = fifo(p1, p2);
```

Listing 7. pFabric with FIFO processing.

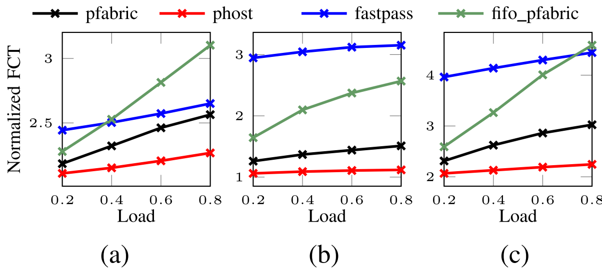


Fig. 4. (a)–(c) Overall average normalized flow completion time for the three workloads with various generated loads according to a Poisson arrival process as in [35].

code changes on control and data planes (cf. DYNAMISM Sect. I).

D. OpenQueue in the Linux Kernel

We have completed a proof-of-concept implementation of OpenQueue [38] in the Traffic Control (TC) layer of the Linux kernel. To that end, we have extended the `tc` Linux command to attach instances of our OpenQueue Queuing Discipline (as a qdisc[4]) to a network interface. Our qdisc is implemented as a Linux kernel module, which can be loaded into the kernel dynamically. An OpenQueue kernel module contains C language constructs corresponding to policy elements. OpenQueue module name is given as a parameter to the `tc` command. For example, if the the module name is `myOpenqueue`, calling the command

```
tc qdisc add dev eth0~root myOpenqueue
```

attaches our qdisc to `eth0` where the OpenQueue policies defined in the input OpenQueue file have been compiled into the module `myOenqueue`. The admission policy is evaluated inside the `enqueue` method. If a packet is admitted, the corresponding processing policy calculates its rank according to processing order. Similarly, the scheduling policy is evaluated inside the `dequeue` method to find the index of the queue that the next HOL packet is taken from. For the priority queue, we use B-Trees that keep pointers to packets (Fig. 5). The operational cost of packet insertions and deletions is $O(log\ N)$, where N is the number of admitted packets.

E. OpenQueue Code Generation for Linux Kernel

Given the abstract nature of OpenQueue syntax and semantic, it is possible to easily generate high-level language code

[4]qdisc is a part of Linux Traffic Control (TC) used to shape traffic of a network interface; qdisc uses `dequeue` to handle outgoing packets and `enqueue` to fetch incoming ones.

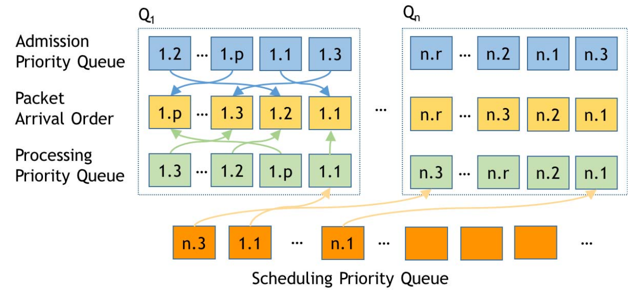


Fig. 5. OpenQueue priority queues in Linux kernel.

for any target runtime environment. We have an OpenQueue language parser and code generation toolset for the Linux kernel, where OpenQueue policies are defined using syntax very similar to Section IV. The *imports* section at the top specifies C header files that define function signatures; they include all possible C routines that can be used as function pointers for various dynamically defined queue/port operations. Imported C routines include the implementations of different OpenQueue actions (e.g., drop or notify) in order to exploit a feature available in the target runtime to simplify language parsing and improve code reusability. These functions are validated for their signatures as each operation has its own specific format and sanity checks. The next section defines queues and their attributes. The last section defines port and its attributes. These three sections collectively define a complete OpenQueue policy. Moreover, queue/port operation attributes are not limited to precompiled C routines but also support a limited set of inline functions for improved usability. The generated code can be compiled into a loadable Linux kernel module that can be attached to a network interface using Linux *tc* command.

F. Priority Queue and Performance

To explore the performance overhead introduced by several priority queues (implemented as B-trees) in OpenQueue, we used priorities based on arrival time to compare it with the base-line qdisc implementation that is implemented as a doubly linked list. We use a 3-node line topology with a sender, receiver and middle node to measure the overhead of our packet prioritization logic. The middle node runs Open vSwitch (OVS) with modified data plane (Linux kernel) and acts as a pass-through switch. We vary the number of parallel traffic generators on the sender and measure average queue length (i.e., number of packets in the default queue) on the receiver for two qdiscs: base-line FIFO and extended FIFO with OpenQueue prioritization; we report the average of 50 runs with 95% confidence interval. Fig. 6(left) shows average queue lengths for the two qdiscs; in both cases, the length increases with the number of UDP clients. In the most congested case, FIFO with 16 clients, regular FIFO has average queue length 559.333 vs. 571 packets for FIFO with prioritization, a mere 2% degradation. We also varied MTU sizes in the same 3-node line topology with 4 parallel UDP generators, which is enough to observe

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

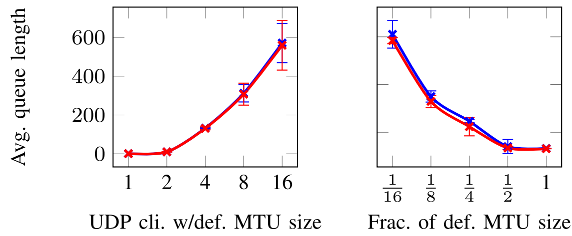


Fig. 6. *Left:* average queue length as a function of number of clients generating UDP traffic with default MTU size. *Right:* fraction of default MTU size; blue: FIFO with prioritization; red: regular FIFO.

queue build-ups without dropping packets in the pass-through switch. We measured average queue lengths of the two qdiscs, varying MTU sizes from $\frac{1}{16}$ of the default MTU size to the default 1500 bytes. Fig. 6(right) shows that for both qdiscs the average queue length decreases as MTU size increases; FIFO with prioritization incurs only 4% overhead: for MTU size of $\frac{1500}{16}$ bytes the result is 584.3 vs. 610.7 packets. This demonstrates that packet prioritization on top of FIFO incurs negligible performance overhead.

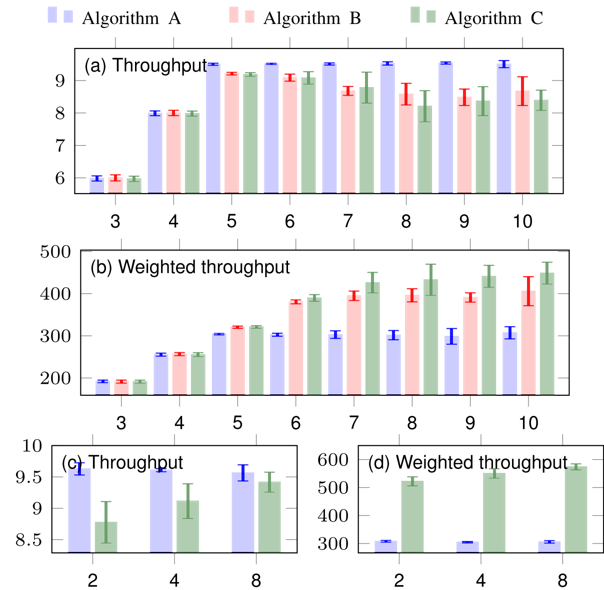


Fig. 7. Average (weighted) throughput (shown on the Y-axis) as a function of input load in Gbps (X-axis) in Experiment 1 (a-b) and no. of queues in Experiment 2 (c-d).

G. Evaluation of Operational Complexity in DPDK

We evaluate the operational complexity of OpenQueue using testbed experiments. We implement OpenQueue using DPDK [39] where the OpenQueue runtime is single-threaded and bound to a dedicated CPU core. During execution, the program polls traffic from a single port assigned to it at initialization. Ingress packets are written to an intermediate ring buffer based on forwarding decisions. Each port has an associated ring buffer, which acts as a thread-safe transmission channel between ingress and egress packet pipelines. The egressing packets are processed according to OpenQueue model where a packet is: (1) admitted based on the admission policy, (2) placed on a queue based on its processing policy, and (3) finally transmitted based on the scheduling policy. We use a setup on CloudLab [40] with 2 source nodes connected to a destination node via a switch node. Each node is a Dell Poweredge R430 machine with two 2.4 GHz 64-bit 8-Core Xeon E5-2630v3 processors, 8 GT/s, 20 MB cache, 64 GB 2133 MT/s DDR4 RAM, and 2 Intel X710 10 GbE NICs. The bandwidth of each interconnection is 10 Gbps. In this setup, the switch node is capable of working as a 10 GbE 3-port switch. We use DPDK version 18.11.2 on Ubuntu 18.04.1 LTS with *igb* as kernel driver. We use *iperf* v. 2.0.13 for our clients and servers. We have UDP clients on the source nodes sending traffic to servers on the destination node depending on the experiment as explained below. We set the load offered by each client (using -b command line option) high enough such that there is congestion, hence queue build up on the switch. On our switch, we record the number of packet drops and packet transmissions and derive average throughput attained by clients to assess the operational overheads incurred by OpenQueue. We use IPv4 Type of Service (ToS) field to mark traffic priority. We use the ToS values 63 and 1 for high-priority and low-priority traffic respectively. The two ToS values represent two distinct traffic priority levels for evaluation of operational complexity versus optimized objectives.

Experiment 1: We compare three algorithms: Algorithm A has FIFO admission and processing priorities and tail drop in case of congestion, Algorithm B also has FIFO admission and processing but drops low-priority packets on congestion, and Algorithm C has FIFO admission priority, SRPT processing priority w.r.t. ToS value, and drops low-priority packets on congestion. Algorithm A is a native implementation without any priority queues (PQs), Algorithm B has one PQ, and (the optimal) Algorithm C has two PQs, for admission and processing. Results of this experiment are shown in Fig. 7a-b. Algorithm C spends the most operations to process a packet, Algorithm B is in the middle, and Algorithm A is the fastest. Thus, Algorithm C can process fewer packets than A and B and as a result, Algorithm A is best in terms of throughput (Fig. 7a). However, due to the smarter processing that more advanced algorithms provide the weighted throughput (total transmitted value) of Algorithms B and C outperforms Algorithm A; in terms of weighted throughput, $A < B < C$ (Fig. 7b). Here, weighted throughput is based on the traffic priority level (high vs. low) to show the advantage of specifically designed algorithms despite additional complexity.

Experiment 2: We assess the impact of using multiple queues attached to the same port. We use $k = 2, 4, 8$ queues equally sharing a total size of 512 with round-robin scheduling, comparing Algorithms A and C defined above; we measure weighted throughput as k changes. As a result (Fig. 7c-d), Algorithm A again outperforms Algorithm C in terms of throughput due to simpler processing but Algorithm C wins very convincingly in weighted throughput.

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

V. Related Work

Frenetic [41], Pyretic [42], among others, focus on service abstractions based on flexible *classifiers*, and do not address management of buffering architectures. Other approaches [43] allow only for a *predefined set* of parameters for buffer management, which intrinsically limits expressivity. While languages such as P4 [11] are very successful in representing packet classifiers, they are less suited to express buffer management policies. Sivaraman *et al.* [25], [44] explored the expression of policies by one priority and one calendar queue, leaving language specification as future work. Mittal *et al.* attempt to build a universal packet scheduling scheme [45]. In contrast to these approaches, OpenQueue considers the composition of admission control, processing, and scheduling policies to optimize chosen objectives on user-defined buffering architectures. New transports such as [7] require complex code changes on control and data planes of network elements to provision desired management policies. Once OpenQueue is supported on the network element, new policies can be added without code changes at runtime.

Although the goal of OpenQueue is to show feasibility even with off-the-shelf priority queue implementations in the most expressive case, to improve operational complexity Eiffel [46] considers a special case with fewer number of prioritization levels, where priorities belong to a restricted range of values. Loom [47] uses a queue per flow and offloads all packet scheduling to NIC. Loom is new NIC design, a different level of abstraction that can exploit OpenQueue/Eiffel abstractions as an internal building block. Recently, Shrivastav [48] considered Push-In-Extract-Out (PIEO) data structure to express buffer management policies. PIEO is more expressive but less efficient than priority queues. SP-PIFO is a programmable packet scheduler abstraction which closely approximates PIFO queues using strict-priority queues [49]. The abstraction for programmable calendar queues is considered in [50]. The event-driven extension of P4 is given in [51]. Most of these abstractions address more specific and lower level scheduling problems compared to OpenQueue (e.g., improved data structures [46], hardware design [48], or integration with existing echo systems such as P4 [51]). Unlike OpenQueue, none of the recent proposals define a clear interface to specify buffer management policies and mostly discuss the expressiveness of a single abstraction. Note that the expressiveness of OpenQueue is defined not only by the priority queue data structure but also several prioritization levels and internal states.

VI. Conclusion

We have proposed a concise yet expressive language to define buffer management policies at runtime; provisioning new buffer management policies does not require control/data-plane code changes. We believe that OpenQueue can enable and accelerate innovation in the domain of buffering architectures and management, similar to programming abstractions that exploit P4 for services with sophisticated classification modules. The complexity of OpenQueue implementation is reduced to efficient support of priority queue data structures. The major contribution of this paper is to develop the right balance between efficiency and feasibility of domain specific language expressing buffering architectures and their management policies. The conciseness of OpenQueue and ability to implement priority queue data structures at line-rate, make OpenQueue attractive for hardware implementations.

References

[1] K. Kogan *et al.*, "A programmable buffer management platform," in *Proc. IEEE 25th Int. Conf. Netw. Protocols (ICNP)*, Oct. 2017, pp. 1–10.

[2] M. H. Goldwasser, "A survey of buffer management policies for packet switches," *ACM SIGACT News*, vol. 41, no. 1, pp. 100–128, Mar. 2010.

[3] S. Nikolenko and K. Kogan, "Single and multiple buffer processing," in *Encyclopedia of Algorithms*. New York, NY, USA: Springer, 2015.

[4] N. Bansal and M. Harchol-Balter, "Analysis of SRPT scheduling: Investigating unfairness," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2001, pp. 279–290.

[5] BBC News. (2014). *US Watchdog to Propose New Net Neutrality Rules*. [Online]. Available: http://www.bbc.com/news/technology-27141121

[6] J. Gettys. (2013). *Low Latency Requires Smart Queuing: Traditional AQM is Not Enough!*. [Online]. Available: http://www.internetsociety.org/sites/default/files/pdf/accepted/29_bis_ISOC_Workshop_2.pdf

[7] M. Alizadeh *et al.*, "PFabric: Minimal near-optimal datacenter transport," in *Proc. ACM SIGCOMM Conf. SIGCOMM (SIGCOMM)*, 2013, pp. 435–446.

[8] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined wan," in *Proc. ACM SIGCOMM Conf. SIGCOMM (SIGCOMM)*, 2013, pp. 3–14.

[9] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM Conf. SIGCOMM (SIGCOMM)*, 2013, pp. 15–26.

[10] N. McKeown *et al.* (2011). *OpenFlow Switch Specification*. [Online]. Available: http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf

[11] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.

[12] A. Kesselman, K. Kogan, and M. Segal, "Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing," *Distrib. Comput.*, vol. 23, no. 3, pp. 163–175, Nov. 2010.

[13] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Netw.*, vol. 1, no. 4, pp. 397–413, Aug. 1993.

[14] W.-C. Feng, K. G. Shin, D. D. Kandlur, and D. Saha, "The BLUE active queue management algorithms," *IEEE/ACM Trans. Netw.*, vol. 10, no. 4, pp. 513–528, Aug. 2002.

[15] K. Nichols and V. Jacobson, "Controlling queue delay," *Commun. ACM*, vol. 55, no. 7, pp. 42–50, Jul. 2012.

[16] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *Proc. Symp. Commun. Archit. Protocols (SIGCOMM)*, 1989, pp. 1–12.

[17] P. E. McKenney, "Stochastic fairness queueing," in *Proc. IEEE INFOCOM, 9th Annu. Joint Conf. IEEE Comput. Commun. Societies Multiple Facets Integr.*, Jun. 1990, pp. 733–740.

[18] W. Aiello, A. Kesselman, and Y. Mansour, "Competitive buffer management for shared-memory switches," *ACM Trans. Algorithms*, vol. 5, no. 1, pp. 1–16, Nov. 2008.

[19] A. Mekkittikul and N. McKeown, "A practical scheduling algorithm to achieve 100% throughput in input-queued switches," in *Proc. IEEE INFOCOM*, Mar./Apr. 1998, pp. 792–799.

[20] A. Kesselman, K. Kogan, and M. Segal, "Improved competitive performance bounds for CIOQ switches," *Algorithmica*, vol. 63, nos. 1–2, pp. 411–424, Jun. 2012.

[21] S.-T. Chuang, S. Lyer, and N. McKeown, "Practical algorithms for performance guarantees in buffered crossbars," in *Proc. IEEE 24th Annu. Joint Conf. IEEE Comput. Commun. Societies*, Mar. 2005, pp. 981–991.

[22] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2016, pp. 501–521.

[23] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

[24] S. Das and R. Sankar. (2012). *Broadcom Smart-Buffer Technology in Data Center Switches for Cost-Effective Performance Scaling of Cloud Applications*. [Online]. Available: https://www.broadcom.com/collateral/etp/SBT-ETP100.pdf

[25] A. Sivaraman *et al.*, "Towards programmable packet scheduling," in *Proc. 14th ACM Workshop Hot Topics Netw. (HotNets)*, 2015, pp. 23:1–23:7.

[26] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal, "Providing performance guarantees in multipass network processors," *IEEE/ACM Trans. Netw.*, vol. 20, no. 6, pp. 1895–1909, Dec. 2012.

[27] P. Chuprikov, S. Nikolenko, and K. Kogan, "Priority queueing with multiple packet characteristics," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2015, pp. 1–9.

[28] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko, "Buffer overflow management in QoS switches," in *Proc. 33rd Annu. ACM Symp. Theory Comput. (STOC)*, 2001, pp. 520–529.

[29] S. Bauer, R. Beverly, and A. Berger, "Measuring the state of ECN readiness in servers, clients, and routers," in *Proc. ACM SIGCOMM Conf. Internet Meas. Conf. (IMC)*, 2011, pp. 171–180.

[30] A. Morton, J. Liu, and I. Song, "Efficient priority-queue data structure for hardware implementation," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2007, pp. 476–479.

[31] A. Ioannou and M. G. H. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *IEEE/ACM Trans. Netw.*, vol. 15, no. 2, pp. 450–461, Apr. 2007.

[32] N. McKeown, "The iSLIP scheduling algorithm for input-queued switches," *IEEE/ACM Trans. Netw.*, vol. 7, no. 2, pp. 188–201, Apr. 1999.

[33] G. Nong, M. Hamdi, and J. K. Muppala, "Performance evaluation of multiple input-queued ATM switches with PIM scheduling under bursty traffic," *IEEE Trans. Commun.*, vol. 49, no. 8, pp. 1329–1333, Aug. 2001.

[34] K. Kogan, A. Lopez-Ortiz, S. I. Nikolenko, and A. V. Sirotkin, "Multi-queued network processors for packets with heterogeneous processing requirements," in *Proc. 5th Int. Conf. Commun. Syst. Netw. (COMSNETS)*, Jan. 2013, pp. 1–10.

[35] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker, "PHost: Distributed near-optimal datacenter transport over commodity network fabric," in *Proc. 11th ACM Conf. Emerg. Netw. Experiments Technol. (CoNEXT)*, 2015, pp. 1–12.

[36] A. Narayan. *Yet Another Packet Simulator (YAPS)*. Accessed: Jul. 28, 2020. [Online]. Available: https://github.com/NetSys/simulator

[37] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized 'zero-queue' datacenter network," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 307–318.

[38] D. Menikkumbura. *Openqueue on Github*. Accessed: Jul. 28, 2020. [Online]. Available: https://github.com/danushkam/openqueueieeetrans

[39] (2019). *Intel DPDK*. [Online]. Available: http://dpdk.org/

[40] D. Duplyakin *et al.*, "The design and operation of CloudLab," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jul. 2019, pp. 1–14.

[41] N. Foster *et al.*, "Frenetic: A network programming language," in *Proc. 16th ACM SIGPLAN Int. Conf. Funct. Program. (ICFP)*, 2011, pp. 279–291.

[42] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 1–13.

[43] R. Soulé *et al.*, "Merlin: A language for provisioning network resources," in *Proc. 10th ACM Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2014, pp. 213–226.

[44] A. Sivaraman *et al.*, "Programmable packet scheduling at line rate," in *Proc. Conf. ACM SIGCOMM Conf. (SIGCOMM)*, 2016, pp. 44–57.

[45] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker, "Universal packet scheduling," in *Proc. 14th ACM Workshop Hot Topics Netw. (HotNets)*, 2015, pp. 24:1–24:7.

[46] A. Saeed *et al.*, "Eiffel: Efficient and flexible software packet scheduling," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2019, pp. 17–32.

[47] B. Stephens, A. Akella, and M. M. Swift, "Loom: Flexible and efficient NIC packet scheduling," in *Proc. 16th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2019, pp. 33–46.

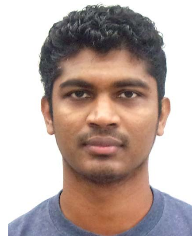
[48] V. Shrivastav, "Fast, scalable, and programmable packet scheduler in hardware," in *Proc. ACM Special Interest Group Data Commun.*, J. Wu and W. Hall, Eds. New York, NY, USA: ACM, 2019, pp. 367–379.

[49] A. G. Alcoz, A. Dietmüller, and L. Vanbever, "SP-PIFO: Approximating push-in first-out behaviors using strict-priority queues," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2020, pp. 59–76.

[50] N. K. Sharma *et al.*, "Programmable calendar queues for high-speed packet scheduling," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2020, pp. 685–699.

[51] S. Ibanez, G. Antichi, G. Brebner, and N. McKeown, "Event-driven packet processing," in *Proc. 18th ACM Workshop Hot Topics Netw.*, Nov. 2019, pp. 133–140.

Kirill Kogan (Member, IEEE) received the Ph.D. degree from the Ben-Gurion University of the Negev, Israel, in 2012. He is a former Technical Leader with Cisco Systems, Inc., where he worked from 2000 to 2012. He was a Post-Doctoral Fellow with the University of Waterloo and Purdue University from 2012 to 2014. He is currently a Senior Lecturer with Ariel University. His current research interests include design, analysis, and implementation of networked systems, broadly defined.



Danushka Menikkumbura received the B.Sc. degree in computer science and engineering from the University of Moratuwa, Sri Lanka. He is currently pursuing the Ph.D. degree in computer science with Purdue University, under the supervision of Prof. Patrick Eugster. He is an Open-Source Software Enthusiast and a Long-Time Contributor and Committer to the Apache Axis and Apache Airavata Projects. His research interests include improving efficiency and robustness of datacenter networks, particularly using the programmability of modern datacenter network hardware.

Gustavo Petri received the Ph.D. degree from the INRIA Sophia Antipolis, Université Nice Sophia Antipolis. He was also a Professor with Paris Diderot University–Paris 7, a Visiting Assistant Professor with Purdue University, and a Post-Doctoral Fellow with DePaul University. He is currently a Security Researcher with Arm Ltd. His research interests include programming languages, concurrency, formal methods, distributed systems, and networking.

Youngtae Noh (Member, IEEE) received the B.S. degree in computer science from Chosun University in 2005, the M.S. degree in information and communication from the Gwangju Institute of Science and Technology (GIST) in 2007, and the Ph.D. degree in computer science from the University of California at Los Angeles (UCLA) in 2012. He is currently an Assistant Professor with the Department of Computer Science and Information Engineering, Inha University. Before joining Inha University, he was a Staff Member with Cisco Systems, Inc., until 2014. His research interests include datacenter networking, wireless networking, future Internet, and mobile/pervasive computing.

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.

Sergey I. Nikolenko received the M.Sc. degree from Saint Petersburg State University in 2005 and the Ph.D. degree from the Steklov Institute of Mathematics, Saint Petersburg, Russia, in 2009. He is currently an Associate Professor with the Higher School of Economics and the Laboratory Head of the Steklov Institute of Mathematics. His research interests include machine learning, analysis of algorithms, and mathematics.

Alexander Sirotkin received the M.Sc. and Ph.D. degrees from Saint Petersburg State University in 2005 and 2011, respectively. He is currently an Associate Professor with the Higher School of Economics at Saint Petersburg, Russia. His research interests include data mining and machine learning, analysis of algorithms, and mathematics.

Patrick Eugster received the Ph.D. degree from EPFL in 2001. He is currently a Professor of computer science with the Università della Svizzera Italiana (USI), Lugano, Switzerland, and a Founding Member of the Computer Systems Institute. He is also an Associate Faculty Member with Purdue University and TU Darmstadt, where he was a Regular Faculty Member before joining USI. His research interests include networked distributed systems, including programmability and protocol design. He received several awards for his research, including the NSF CAREER Award in 2007 and the ERC Consolidator in 2014.

Authorized licensed use limited to: Korea Institute of Energy Technology (KENTECH). Downloaded on March 14,2023 at 11:47:38 UTC from IEEE Xplore. Restrictions apply.