A Programmable Buffer Management Platform

Kirill Kogan*, Danushka Menikkumbura†, Gustavo Petri‡, Yangtae Noh§, Sergey Nikolenko¶, Alexander Sirotkin‖, Patrick Eugster†**

*IMDEA Networks Institute †Purdue University ‡Paris Diderot - Paris 7 §Inha University **TU Darmstadt
¶Steklov Institute of Mathematics at St. Petersburg ‖National Research University Higher School of Economics

***Abstract*—Buffering architectures and policies for their efficient management constitute one of the core ingredients of a network architecture. However, despite strong incentives to experiment with, and deploy, new policies, the opportunities for alterating anything beyond minor elements of such policies are limited. In this work we introduce a new specification language, OpenQueue, that 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 direct impact on performance in various settings.**

I. Introduction

Buffering architectures define how input and output ports of a network element are connected [13]. 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* [3] and the consideration of 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 [22]) which do not really capture buffer management aspects. This calls for novel 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; (4) DYNAMISM: new policies should be feasible to specify at run-time without any code changes and (re-)deployments.

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 these primitives. The choice of primitives dictates the SIMPLICITY and EXPRESSIVITY of the language. Abstractions of ordered sets of conditions and actions governing the handling of individual incoming packets as in OpenFlow can be too complex and inefficient for a language for buffer management, thus motivating OpenQueue's primitives.

OpenQueue has two main types of objects: *ports* and *queues* assigned to ports. Each queue has an *admission control policy*, determining which packets are admitted or dropped [11], [10], [26]. Each port defines a *scheduling policy*, used to select a queue whose HOL packet will be processed next [9], [21]; in each queue is defined by a *processing policy*. In some cases, e.g., shared memory switches [2], several queues share the same *buffer space*, and the admission control policy can routinely query the state of several queues; for instance, the LQD policy under congestion drops packets from the longest queue [2]. To capture these architectures, OpenQueue deals with buffers and admission control policy to resolve congestion at the buffer level. 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) [23], [18] or asynchronously, like packet scheduling in a buffered crossbar switch [7], [17], specific *implementations* are beyond the scope of OpenQueue.

In summary, defining a buffering architecture and its management in OpenQueue requires creating instances of ports, queues, and buffers, and specifying relations among them: admission control, processing, and scheduling policies attached to these instances.

A. 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 [24], which explores specifications of universal scheduling policies that can satisfy multiple objectives instead of having a flexible interface to define new buffer management policies (and later studies its combinations with admission control), we argue in the following 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 [5], 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 algorithm that discarded the first packet and is now able to accept all new arrivals. □

Consider a different policy for SQ that does not conform to FIFO order, processing the most valuable packet first and in case of congestion pushing out the least valuable packets. We call it PQ_1; it is clearly an online policy.

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

Proof. First, note that in case of packets with the same processing time of one timeslot per packet, 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 but with each queue having FIFO processing and dedicated to packets with the same value (several queues can have the same value). This simplifies processing and moves complexity to scheduling decisions. Clearly, the algorithm that pushes out the globally least valuable packet and schedules the most valuable packet transmits the same total value as PQ_1 and the following immediately follows.

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 [8]. 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, based on this information it is possible to recreate the processing order in SQ if there are no new arrivals. This computed processing order can be used as an implemented priority function in the SQ architecture. Basically, 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 probabilistic case, the behaviour of policies in each specific case may diverge 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, both with deterministic policies and probabilistic ones. So do we really need multiple queues on a single output port (assuming we have no physical constraints such as memory access bandwidth)? In fact, Sivaraman et al. [28] recently proposed to express policies with a single priority queue and a single calendar queue. However, even though both architectures are equally expressive, 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 consider 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 the 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 in our architecture; 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.

The lower bound stems from the fact that by adding one or two packets we can completely reorder the single queue that emulates LQF.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 $p_{1,2}p_{2,1}\cdots p_{m,1}p_{1,3}p_{2,2}\cdots p_{m,2}p_{3,1}\cdots p_{m,m}p_{1,m+2}$, 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. $\square$

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 *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*.

III. OpenQueue Specification Language

Next we present the abstractions provided by OpenQueue, aiming to reconcile SIMPLICITY and EXPRESSIVITY while keeping PERFORMANCE in mind (cf. Sec. I). We begin with some common abstractions used to declare the primitives manipulated by OpenQueue.

A. Comparators

The core data structure for the expression of buffering architectures is the *priority queue*. 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 (over different types). Since comparators are used internally by OpenQueue to implement the queues, the comparison operation has to be efficiently computable (ideally at the hardware level). To achieve efficient computation with comparators, OpenQueue imposes certain syntactic restrictions on their definition. The syntax below captures the main restrictions.

$$\begin{array}{lcll}
x & & & \text{formal variables} \\
n & & & \text{numeric constants} \\
e & ::= & n \mid x.f \mid e \oplus e & \text{arithmetic expressions} \\
b & ::= & e \olessthan e \mid b \owedge b & \text{predicates} \\
c & ::= & comp\_name(x, x) = b & \text{comparator def.}
\end{array}$$

Comparator declarations require providing a name *comp_name*, and take two arguments. The type of the arguments varies depending on the priority being defined. For instance for queues in OpenQueue, which hold packets, 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 $\owedge$) and arithmetic relations (denoted with $\olessthan$), 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 the 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.

B. Boundary Conditions

Data structures manipulated by OpenQueue can behave differently depending on whether the network entity is operating in *normal state*, or in *congested state*. For example, when a queue or a buffer becomes saturated, the user could specify that certain packets should be dropped to achieve graceful degradation. We allow the user to specify conditions under which a data structure should be considered congested. 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.

$$\begin{array}{lcll}
pf & ::= & \texttt{weightAdm} \mid \texttt{weightSched} & \text{modifiables} \\
ac & ::= & \mathbf{\texttt{drop}}(P) & \text{actions} \\
 & \mid & \mathbf{\texttt{modify}}(pf := e) \mid \mathbf{\texttt{mark}} \mid \mathbf{\texttt{notify}} \mid ac \cdot ac & \\
cl & ::= & (b,\ ac) \mid (b,\ ac) \cdot cl & \text{condition cases} \\
cd & ::= & cond\_name(x) = cl & \text{declarations}
\end{array}$$

This syntax shares the definitions of predicates and declarations with comparators seen before. Importantly, boundary conditions 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 predicate and second component defines an *action* represented by the meta-variable ac.[1] For the time being, we focus on the **drop**(P) action which indicates that packets have to be

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

dropped from the queue with probability P if the matching predicate evaluates to true. Actions **modify**($pf := e$), **mark**, and **notify** will be discussed later. Moreover, we allow actions to be sequenced, although generally only one action is used in conditions. Condition cases enable the expression of different response scenarios according to different types of congestion. 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 sequel we present the different entities comprising OpenQueue in detail. For each entity we provide its properties; some are primitives of the domain (e.g., packet size), and others have to be set by the programmer. For each property we indicate in comments whether it is **r** read-only or **rw** writable, and **cons** if it's value is fixed during execution, or **dyn** otherwise. For 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**).

C. Queues

List. 1 shows the declaration of queues as first ingredient of the clearly defined core OpenQueue programming interface. The standard property `size` is defined by the user at declaration time. The same goes for the `buffer`, which represents the buffer that contains the queue, and in the case of shared memory is shared among several queues. The `currSize` property serves to query the current size and changes dynamically as the queue is updated. Abstractly, a queue contains packets ordered according to user-defined priorities for admission control and processing.

a) Admission Policy: The first policy concerns the admission of packets into the queue: `admPrio(p1, p2)` is a packet comparator used in case of congestion to choose the packets to be dropped from the queue. As an alternative, instead of defining an admission policy we could simply drop the least valuable packets according to `procPrio` priority that we will describe shortly. However, as shown in Sec. IV, separate priorities for admission and processing not only give more EXPRESSIVITY but also improve PERFORMANCE. There are several properties related to the admission policy. 1) The user-defined `congestion()` boundary condition that shows when a queue is virtually *congested*, and defines which/how packets should be dropped (here we only consider the **drop**(P) action). The single argument of boundary condition declarations is implicitly instantiated to the queue being defined, hence this is a queue boundary condition. We notice here that the deterministic drop action corresponds to an action **drop**(1) in the syntax presented before. Usually, `congestion()` is a set of different buffer occupancies and drop probabilities [11]. In the example below we show a possible congestion policy whereby packets start being dropped with a probability of .5 if the current occupation of the queue is greater than 3/4 of the total size of the queue but lower than 9/10; they are dropped with a probability of .9 if the occupation excedes 9/10 but is lower than 19/20; and they are always dropped otherwise.

```
congestion() = |
   (currSize >= .95*size, drop(1) ) .|
      (currSize >= .9*size, drop(.9) ) .|
         (currSize >= .75*size, drop(.5))|
```

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. 2) The optional function `postAdmAct()` is a boundary condition like `congestion()`, except that it is restricted to **mark**, **notify**,

```
Queue {
 // user-specified at declaration
 size             // size in bytes     [r, cons]
 buffer           // allocating buffer [r, cons]
 // primitive properties
 currSize         // current size       [r, dyn]
 // admission -- user-specified at decl.
 admPrio(p1, p2)  // pushOut comparat.[bool fun]
 congestion()     // drop(P) condit. [drop cond]
 postAdmAct()     // [{mark,notify,modify} comp]
 weightAdm        // adm. priority     [rw, dyn]
 // processing -- user-specified at decl.
 procPrio(p1, p2)// proc comparat.[Packet comp]
 getHOL()         // HOL packet     [Packet fun]
 // scheduling -- user-specified at decl.
 weightSched      // scheduling prio.  [rw, dyn]
}

Buffer {
 // primitive properties
 currSize          // current size       [r, dyn]
 getBestQueue()    // on weightAdm   [Queue fun]
 getCurrQueue()    // admitted one   [Queue fun]
 // user-specified at declaration
 size              // size              [r, cons]
 // admission -- user-specified at decl.
 congestion()      // drop(P)         [drop cond]
 queuePrio(q1, q2)// compare q-s     [bool fun]
 postAdmAct()      //[{mark,notify,modify} cond]
}

Port {
 // primitive properties
 getBestQueue()    // on weightSched [Queue fun]
 getCurrQueue()    // scheduled one  [Queue fun]
 // scheduling user-specified at decl.
 schedPrio(q1, q2)// compare q-s     [bool fun]
 postSchedAct()    //[{mark,notify,modify} cond]
}

Packet {
 size               // size in bytes   [r, cons]
 value              // virtual value   [r, cons]
 processing         // nb of cycles    [r,  dyn]
 arrival            // arrival time    [r, cons]
 slack              // offset in time  [r, cons]
 queue              // target queue id [r, cons]
 flow               // flow id         [r, cons]
}
```

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

and **modify**$(pf := e)$. These actions are intended to tell subsequent processing entities that the packet is subject to special conditions. 3) Finally, the function `postAdmAct()` can be used to implement *explicit congestion notifications* [4] or *backpressure*; `postAdmAct()` can return actions such as **mark** or **notify**. When bandwidth is allocated not only with respect to packet attributes, queues maintain a `weightAdm` variable that can be updated dynamically after each scheduling, and the is what **modify**$(pf := e)$ is for.

b) Processing Policy: The processing policy defines the priorities of packets in the queue through `procPrio(p1, p2)`: a packet comparator defined as a function taking two abstract packets and returning *true* if `p1` has a higher processing priority than `p2`. We are only concerned with the highest processing priority packet at any point. This priority defines the most and least valuable packets in the queue. Hence, the only way to access packets in the queue ordered by `procPrio` is through the `getHOL()` primitive which returns the HOL (i.e., packet with highest processing priority as defined by `procPrio`). As an example, the user can set

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

to encode FIFO processing so calls to `getHOL()` return the packet with oldest arrival time.

c) Scheduling Policy.: This policy allows to specify static bandwidth allocations among queues of the same port during scheduling. In this case, the whole policy is defined in part in the queue declaration, and in part in the port declarations. The `weightSched` variable of each queue can be updated by the `postSchedAct()` function defined in ports to that end as we shall see shortly.

D. Buffers

A buffer is an optional entity, declared only when several queues share buffer space (see List. 1). It manages a set of queues assigned to it at creation; `congestion()`, `postAdmAct()`, `size`, and `currSize` are similar to the respective queue attributes. Under congestion, an admission control policy on the buffer level finds a queue whose packet should be dropped, and the queue's admission control policy determines which packet to drop. To order queues for admission, the user specifies the `queuePrio` comparator. For example, to implement LQD the following comparator can be used:

```
queuePrio(q1, q2) = (q1.currSize < q2.currSize)
```

E. Ports

The interface for ports is presented in List. 1. A port manages a set of queues assigned to it at its declaration.[2] `schedPrio(q1,q2)` is a user-defined scheduling property that defines which HOL packet is scheduled next (this packet is accessed through `getBestQueue()`). For example, priority

[2]We leave the `new` operator used to create network objects in OpenQueue implicit; its usage will be clear from examples in Sec. IV.

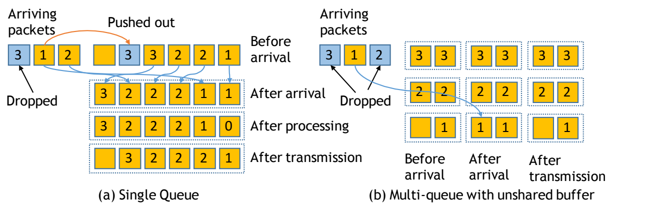


Fig. 1. Packets with required processing. Left: single priority queue with buffer of size $B = 6$; right: multiple separated queues with three queues ($k = 3$) of size 2 each. Dashed lines enclose queues.

based on packet values with several levels of strict priorities is defined as

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

Finally, `postSchedAct()` is similar to `postAdmAct()` and is used to define new services.

F. Packets

The notion of a packet is *primitive*, meaning that the user cannot modify or extend packets; packet fields can be used to implement policies. To be independent of traffic types and to have a clear separation from the classification module (that can be expressed in a different language), every incoming packet is prepended with three mandatory parameters, *arrival time*, *size* in bytes, and *destination queue*, and four optional parameters, intrinsic *value* (with application-specific meaning), *processing requirement* in virtual cycles, *slack* (maximal offset in time from *arrival* to transmission), and *flow* (a traffic aggregation that the packet belongs to). We assume these properties are set by an external *classification unit* (e.g., OpenFlow [22], if a virtual switch is defined with the finest possible resolution), except for *arrival* (set by OpenQueue when a packet is received) and *size*.

List. 1 depicts the **Packet** data structure. Intrinsic *value* and *processing* requirements are used to define prioritization levels [16]. *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; see Sec. 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 (e.g., buffer occupancy).

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.

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. 1(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.[3] List. 3 shows the full specification of a SQ buffering architecture and its optimal throughput policy.

```
// priorities for admission and processing
fifo(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.
defCongestion() =
  lambda q, (q.currSize >= q.size, drop(1))
```

Listing 2. Example priorities and congestion conditions.

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

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

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 [16], [27]; OPT/ALG is the competitive ratio between the throughput of an optimal offline (OPT) algorithm and an online algorithm (ALG). 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:

[3]We use **`lambda`** and **`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.

```
q1.admPrio = fifo;
q1.procPrio = fifo;
```

Changing `fifo()` admission order to `rsrpt()` significantly improves performance and this version of the greedy policy 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 [16]. Since here a port manages only one queue, a *scheduling policy* is just an implicit call to `getHOL()`.

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

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

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. 1(b)). The OpenQueue code below 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 [19], [27]. 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 [19]. 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 snippet in OpenQueue, for instance, corresponds to the CRR policy:

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

TABLE I
SAMPLE OPENQUEUE POLICIES WITH ANALYTIC RESULTS FOR SQ ARCHITECTURE; k – MAXIMAL PROCESSING REQUIREMENT PER PACKET IN CYCLES.

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

TABLE II
SAMPLE POLICIES IN OPENQUEUE FOR MULTIPLE SEPARATED QUEUES ARCHITECTURE.

```
// 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.weightSched > q2.weightSched);
// MINQF: HOL packet from queue that admits
// min processing
minqf(q1,q2) = (q1.weightSched < q2.weightSched);
// CRR: Round-Robin with per cycle resolution
crr(q1,q2) = (q1.weightSched < q2.weightSched);
crrPostSchedAct() =
 lambda port,
  let q = port.getCurrQueue() in
   (true, // condition
    modify(q.weightSched := q.weightSched+k));
// PRR: Round-Robin with per packet resolution
prr(q1,q2) = (q1.weightSched < q2.weightSched);
prrPostSchedAct() =
 lambda port,
  (let q = port.getCurrQueue() in
    (q.getHOL().processing == 0, // condition
     modify(weightSched := weightSched+k*k)));
```

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

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

Listing 6. CRR policy for multiple separated queues.

```
//Create n queues of size B
q1 = Queue(B); ...; qn = Queue(B);
//Create a shared buffer of size B
// and attach queues to it
b = Buffer(B, q1, ..., qn);
//Create an output port per queue
out1 = Port(q1); ...; outn = Port(qn);
```

Listing 7. Shared memory switch architecture.

```
//fifo admission order for queues
q1.admPrio = fifo; ...; qn.admPrio = fifo;
//fifo processing order
q1.procPrio = fifo; ...; qn.procPrio = fifo;
//congestion condition
q1.congestion = defCongestion(q1); ...;
  qn.congestion = defCongestion(qn);
//buffer admission priority
queuePrio(q1, q2) = (q1.currSize<q2.currSize)
b.admPrio = queuePrio;
//congestion condition
defCongestionBuf() =
  (b.currSize>=b.size, drop(1))
b.congestion = defCongestionBuf(b);
```

Listing 8. Longest-Queue-Drop (LQD) policy for shared memory switch.

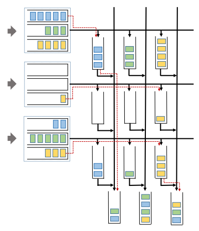


Fig. 2. Buffered-crossbar switch with three hierarchical levels.

C. Admission in Shared Buffers

Next we consider a shared memory switch architecture [8]. Listing 7 exemplies it in OpenQueue. As a sample policy we declare an LQD policy, which works very well for throughput maximization regardless of traffic patterns [2]. LQD greedily accepts all packets and upon congestion drops a packet from the currently longest queue. LQD is at least $\sqrt{2}$ and at most 2-competitive [2] versus an optimal offline policy; Listing 8 shows a complete definition of LQD.

D. Buffered Crossbar Switch

In this subsection we show an example of multi-level buffering architecture that demonstrates the applicability of OpenQueue to specify management policies for switching fabrics. In difference from an input-queued or combined-input-output-queued switch that requires synchronous policies that usually compute matching between input and output ports, adding an additional buffering level at crosspoints allows to make this buffering architecture asynchronous. Here, we consider a full-fledged version with three buffering levels, where the first level implements virtual-output queues (see Fig 2 and Listing 9). Listing 10 shows a policy with longest-queue-first on input and output ports.

E. Software-Defined Transports

Recently, new transports were introduced to optimize various objectives (throughput, average flow completion time, etc.) [3], [12]. Some of them require complex processing orders and support of push-out whose incorporation can require complex code changes both on control and data planes. E.g., pFabric [3] prioritizes packets according to remaining flow completion time (FCT) and during congestion pushes out least valuable packets. In the original implementation of pFabric, whenever dequeuing a packet a linear search over the entire queue finds the first (top priority) packet from the flow in order not to reorder packets in the same flow. Although [3] mentions that evaluations do not encounter long queues sizes, on general workloads this can become a bottleneck. Since pFabric was

```
// create 9 virtual-output queues
voq11 = Queue(B); ...; voq33 = Queue(B);
// attach voqs to input ports
in1 = Port(voq11, voq12, voq13);
in2 = Port(voq21, voq22, voq23);
in3 = Port(voq31, voq32, voq33);
// create 9 crosspoint queues of size 1
// usually small buffers are enough
cq11 = Queue(1); ...; cq33 = Queue(1)
// crosspoints as ports
cp11 = Port(cq11); ...; cp33 = Port(cq33);
// create 3 output queues of size B
oq1 = Queue(B); oq3 = Queue(B);
// attach oqs to output ports
out1 = Port(oq1); ...; out3 = Port(oq3);
```

Listing 9. Specification of 3x3 Buffered-crossbar switch.

```
// setting queues:
// admission order to fifo
voq11.admPrio = fifo; ...; voq33.admPrio = fifo;
cq11.admPrio = fifo; ...; cq33.admPrio = fifo;
oq1.admPrio = fifo; ...; oq3.admPrio = fifo;
// processing order to fifo
voq11.proPrio = fifo; ...; voq33.proPrio = fifo;
cq11.proPrio = fifo; ...; cq33.proPrio = fifo;
oq1.proPrio = fifo; ...; oq3.proPrio = fifo;
// congestion condition
voq11.congestion = defCongestion(); ...;
cq11.congestion = defCongestion(); ...;
oq1.congestion = defCongestion(); ...;
// LQF: HOL pkt. from Longest-Queue-First
lqf(q1, q2)  = (q1.currSize > q2.currSize);
in1.schedPrio = lqf; in2.sched.Prio = lqf;
out1.schedPrio = lqf; out2.schedPrio = lqf;
```

Listing 10. Longest-Queue-First on input and output ports.

```
// buffering architecture specification
q1 = Queue(B);
out = Port(q1);
// admission control
q1.admPrio(p1, p2) = (p1.value > p2.value);
q1.congestion = defCongestion(q1);
// processing policy: fifo()
q1.admPrio(p1, p2) = fifo(p1, p2);
```

Listing 11. pFabric with FIFO processing.

evaluated in the discrete simulator YAPS [25], 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 still push out the least valuable packet during congestion but process packets in FIFO order. Listing 11 shows this new transport in OpenQueue.

In the simulation environment, pFabric with FIFO clearly cannot be better than pFabric that processes packets with shortest remaining flow size first. But even here, normalized FCT[4] is close at least on the IMC10 workload (see Fig. 3). Our goal here is not to introduce another transport but to show how easily OpenQueue can introduce new transports at run-time without any code changes on control and data planes.

[4]Normalized flow completion time is the ratio of mean FCT(i) and mean OPT(i), where OPT(i) is the completion time of flow i when it is the only flow in the network, and FCT(i) is the actual completion time [12], [15].

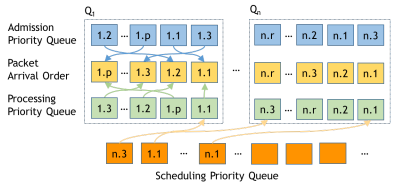


Fig. 4. OpenQueue priority queues in Linux kernel.

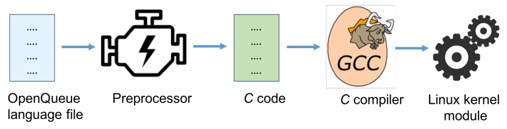


Fig. 5. From OpenQueue language to Linux kernel module.

V. Feasibility of OpenQueue

A fundamental building block in OpenQueue is the *priority queue* data structure, where the order of elements is maintained based on a user-defined priority. Our implementation keeps a single copy of each packet and uses pointers to encode priorities (see Fig. 4). Therefore, the performance of Open-Queue on a given platform largely boils down to the efficiency of underlying priority queue implementation. While priority queue operations take $O(\log N)$ time in general, where N is a queue size, there are restricted versions (e.g., for predefined ranges of priorities) that support most operations in $O(1)$ and can be efficiently implemented even in hardware [14], [28], further increasing OpenQueue's appeal. 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. The complexity of OpenQueue is hence reduced to translating user-defined settings to a target system that implements a virtual buffering architecture.

A. OpenQueue in the Linux Kernel

We implemented OpenQueue [1] in the Traffic Control (TC) layer of the Linux kernel. To that end, we have extended the Linux command `tc` to attach instances of OpenQueue Queuing Discipline (as a qdisc[5]) to a network interface. Our qdisc is implemented as a Linux kernel module, which can be loaded into the kernel dynamically. A OpenQueue kernel module contains C language constructs correspond to OpenQueue policy elements. OpenQueue module name is given as a parameter to the `tc` command.

Fig. 5 illustrates how the loadable OpenQueue policy modules are generated. The input of our preprocessor is an OpenQueue file containing the desired architecture for the interface. Then, the preprocessor generates a corresponding C source code for that. Subsequently, we compile this file into a loadable kernel module to be loaded when using the `tc` command dynamically (cf. the `insmod` command). This

[5]qdisc is a part of Linux Traffic Control used to shape traffic on an interface; it uses `dequeue` for outgoing packets and `enqueue` for incoming ones.

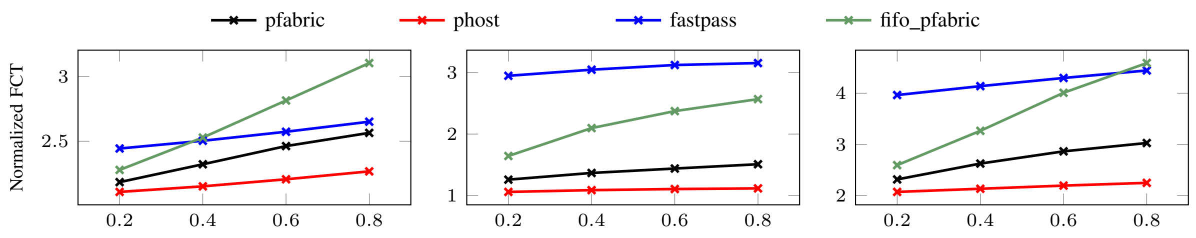
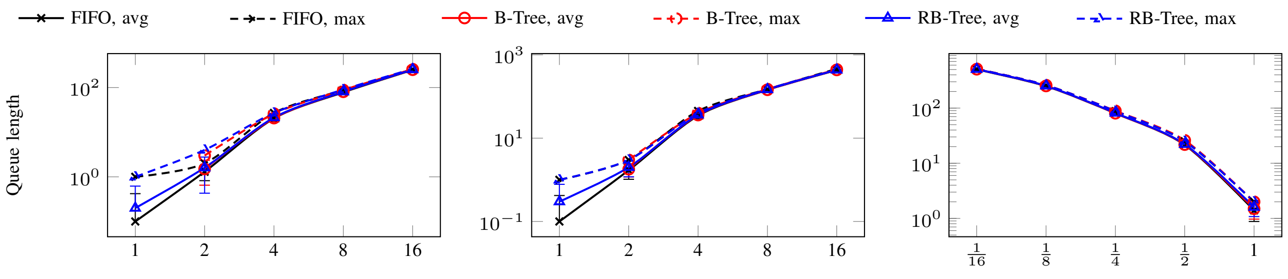


Fig. 3. Overall average normalized flow completion time for the three workloads with various loads as in [12].



(a) UDP clients with default MTU size (b) UDP clients with half default MTU size (c) Fraction of default MTU size

Fig. 7. Queue length as a function of (a) number of UDP clients with default MTU size; (b) with half default MTU size; (c) fraction of default MTU size.

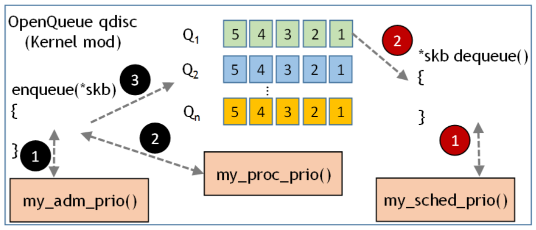


Fig. 6. Use of function calls inside the kernel module during packet `enqueue` and `dequeue`.

allows to load new policies seamlessly, without disrupting currently executing policies on other interfaces. 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. This interaction between the qdisc and predefined functions is depicted in Fig. 6. As a priority queue data structure we use B-Trees and RB-trees that keep pointers to packets (Fig. 4). The operational cost of packet insertions and deletions is $O(log\ N)$, where N is number of admitted packets.

B. OpenQueue Code Generation for Linux Kernel

Given the abstract nature of OpenQueue syntax and semantic, one can generate high-level language code for any target runtime environment. We developed an OpenQueue language parser and code generation toolset for the Linux kernel. A sample policy file in Listing 12 has three main sections. First, imports at the top specify C header files that define function signatures, including all possible C routines that can be used as function pointers for dynamically defined queue/port operations. These functions' signatures are validated for the format of each operation and mundane sanity checks: argument types, duplicate function names, missing semicolons, etc. The next section defines queues and their attributes. The last section defines the port and its attributes. These three sections define a complete OpenQueue policy. Queue/port operation attributes are not limited to precompiled C routines but also support a limited set of inline functions for improved usability. The code can be compiled into a loadable Linux kernel module attached to a network interface using Linux *tc* command.

```
import "include/routine/routines.h"
// Create queues with packets of size 128 and 1024
Queue q1 = Queue(128);
Queue q2 = Queue(1024);
// Attributes of q1
q1.admPrio = my_adm_prio;
q1.congestion = my_congestion_condition;
q1.congAction = drop_tail;
q1.procPrio = my_pro_prio;
// Attributes of q2
// TOS field as admission priority
q2.admPrio = inline{Packet.TOS};
// Queue is congested if its length is 1024 packets
q2.congestion = inline{Queue.length == 1024};
// Drop packets with 95% prob. when congested
q2.congAction = drop_tail(0.95);
q2.procPrio = my_pro_prio;
// Create port
Port myPort = Port(q1, q2);
// Define port attributes
myPort.queueSelect = select_admission_queue;
myPort.schedPrio = my_schd_prio;
```

Listing 12. Sample policy file.

C. Priority Queue and Performance

To explore the performance overhead introduced by priority queues (implemented as B-trees or RB-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 used a testbed with a 3-node line topology to measure the performance 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 first node and measure average queue length (i.e., number of packets in the default queue) and the maximal queue size on the third receiver node for three qdiscs: base-line FIFO and FIFO with B-tree and RB-tree prioritization in OpenQueue, reporting the average value of 50 runs with 95% confidence interval. Fig. 7(a-b) shows the average queue lengths and maximal queue lengths for the three qdiscs; in all cases, average queue length increases with the number of UDP clients. In FIFO with 16 clients, the most congested case, regular FIFO has an average queue length 247.8 vs. 251.1 packets for FIFO with prioritization, a mere 1.3% degradation. We also varied MTU sizes in the same 3-node line topology with 4 parallel UDP generators, which is enough to observe queue build-ups without dropping packets in the pass-though switch. We measured average and maximal queue lengths of the three qdiscs by varying MTU sizes from $\frac{1}{16}$ of the default MTU size to its default size (1500 bytes). Fig. 7(c) shows that for both qdiscs average queue length decreases as MTU size increases; FIFO with prioritization has about 1% overhead, both on average and in maximal values: for MTU size of $\frac{1500}{16}$ bytes the averages are 501.6 vs. 506.7 packets. This demonstrates that packet prioritization incurs negligible performance overhead.

VI. Related Work

Languages such as P4 [6] are very successful in representing packet classifiers, but they are less suited to express buffer management policies. Our work was inspired by [28] that introduces a set of primitives to define admission control policies. Recently, Sivaraman et al. [28], [29] expressed policies by one priority and one calendar queue, still leaving the language specification as future work. Mittal et al. attempt to build a universal packet scheduling scheme [24]. In contrast to these approaches, OpenQueue considers a composition of admission control, processing, and scheduling policies to optimize chosen objectives on user-defined buffering architectures. Some preliminary thoughts leading to the design of OpenQueue have been presented in a short paper [20].

VII. Conclusions

We have proposed a concise yet expressive language to define buffer management policies at runtime; new buffer management policies do not require control/data-plane code changes. We believe that OpenQueue can enable and accelerate innovation in buffering architectures and management, similar to programming abstractions that exploit OpenFlow for services with sophisticated classification modules.

Acknowledgments: K. Kogan was partially supported by Ministerio de Economia y Competitividad grant TEC2014-55713-R. Y. Noh was supported in part by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2016R1C1B2011415). P. Eugster was supported by ERC Consolidator grant #617805 (LiveSoft) and DFG Center #1053 (MAKI). D. Menikkumbura was supported by NSF under grant #1117065.

References

[1] Openqueue on github. https://github.com/openqueuenew/icnp.
[2] W. Aiello and et al. Competitive buffer management for shared-memory switches. *ACM Trans. on Algorithms*, 5(1), 2008.
[3] M. Alizadeh and et al. pfabric: minimal near-optimal datacenter transport. In *SIGCOMM*, pages 435–446, 2013.
[4] S. Bauer and et al. Measuring the state of ECN readiness in servers, clients, and routers. In *IMC*, pages 171–180, 2011.
[5] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
[6] P. Bosshart and et al. P4: programming protocol-independent packet processors. *CCR*, 44(3):87–95, 2014.
[7] S. Chuang, S. Iyer, and N. McKeown. Practical algorithms for performance guarantees in buffered crossbars. In *INFOCOM*, pages 981–991, 2005.
[8] S. Das and R. Sankar. Broadcom smart-buffer technology in data center switches for cost-effective performance scaling of cloud applications, 2012. https://www.broadcom.com/collateral/etp/SBT-ETP100.pdf.
[9] A. Demers and et al. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, pages 1–12, 1989.
[10] W. Feng and et al. The BLUE active queue management algorithms. *IEEE/ACM Trans. Netw.*, 10(4):513–528, 2002.
[11] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.
[12] P. Gao and et al. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT*, pages 1–12, 2015.
[13] M. Goldwasser. A survey of buffer management policies for packet switches. *SIGACT News*, 41(1):100–128, 2010.
[14] A. Ioannou and M. Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Trans. Netw.*, 15(2):450–461, 2007.
[15] P. Jonathan and et al. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM*, pages 307–318, 2014.
[16] I. Keslassy and et al. Providing performance guarantees in multipass network processors. *IEEE/ACM Trans. Netw.*, 20(6):1895–1909, 2012.
[17] A. Kesselman and et al. Packet mode and qos algorithms for buffered crossbar switches with FIFO queuing. *Distributed Computing*, 23(3):163–175, 2010.
[18] A. Kesselman and et al. Improved competitive performance bounds for CIOQ switches. *Algorithmica*, 63(1-2):411–424, 2012.
[19] K. Kogan and et al. Multi-queued network processors for packets with heterogeneous processing requirements. In *COMSNETS*, pages 1–10, 2013.
[20] K. Kogan and et al. BASEL (buffer management specification language). In *ANCS*, pages 69–74, 2016.
[21] P. McKenney. Stochastic fairness queueing. In *INFOCOM*, pages 733–740, 1990.
[22] N. McKeown and et al. OpenFlow switch specification, 2011. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.
[23] A. Mekkittikul and N. McKeown. A practical scheduling algorithm to achieve 100% throughput in input-queued switches. In *INFOCOM*, pages 792–799, 1998.
[24] R. Mittal and et al. Universal packet scheduling. In *NSDI*, pages 501–521, 2016.
[25] A. Narayan. Yet another packet simulator (YAPS). https://github.com/NetSys/simulator.
[26] K. M. Nichols and V. Jacobson. Controlling queue delay. *Commun. ACM*, 55(7):42–50, 2012.
[27] S. Nikolenko and K. Kogan. Single and multiple buffer processing. In *Encyclopedia of Algorithms*. Springer, 2015.
[28] A. Sivaraman, , and et al. Towards programmable packet scheduling. In *HotNets*, pages 23:1–23:7, 2015.
[29] A. Sivaraman and et al. Programmable packet scheduling at line rate. In *SIGCOMM*, pages 44–57, 2016.