BASEL (Buffer mAnagement SpEcification Language)

Kirill Kogan
IMDEA Networks Institute
kirill.kogan@imdea.org

Danushka Menikkumbura
Purdue University
dmenikku@purdue.edu

Gustavo Petri
Université Paris Diderot - Paris 7
gpetri@liafa.univ-paris-diderot.fr

Youngtae Noh
Inha University
ytnoh@inha.ac.kr

Sergey Nikolenko
Steklov Math. Institute at St. Petersburg
National Research University Higher School of Economics
sergey@logic.pdmi.ras.ru

Patrick Eugster
Purdue University
TU Darmstadt
p@cs.purdue.edu

Abstract

Buffering architectures and policies for their efficient management constitute one of the core ingredients of a network architecture. In this work we introduce a new specification language, BASEL, that allows to express virtual buffering architectures and management policies representing a variety of economic models. BASEL does not require the user to implement policies in a high-level language; rather, the entire buffering architecture and its policy are reduced to several comparators and simple functions. We show examples of buffer management policies in BASEL and demonstrate empirically the impact of various settings on performance.

1. INTRODUCTION

Buffering architectures define how input and output ports of a network element are connected [17, 36]. Their design and management must thus be done with care, as it directly impacts 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* and the consideration of additional traffic properties lead to new challenges in the implementation and performance for traditional switching architectures [16, 18, 20]. Unfortunately, current developments in software-defined networking mostly sidestep these challenges by concentrating on flexible and efficient representations of *packet classifiers* (e.g., OpenFlow [32]) which do not really capture buffer management aspects. This calls for novel abstractions that enable the definition of buffer management policies that can be deployed on real network elements at runtime (without respin of implementing hardware and complex code changes). Designing such abstractions however is non-trivial, as they must satisfy a number of possibly conflicting requirements: (1) EXPRESSIVITY: expressible policies should cover various buffering architectures representing 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: the implementations of policies should be efficient on "virtual switches", that is with various resolutions ranging from a single network element to the whole network (e.g., an interconnect for geographically distributed data centers [18, 20]).

We address these challenges with BASEL (*Buffer mAnagement SpEcification Language*), a flexible way to define buffer management policies.

ANCS '16, March 17-18, 2016, Santa Clara, CA, USA

DOI: http://dx.doi.org/10.1145/2881025.2881027

2. BASEL SPECIFICATION LANGUAGE

2.1 Language Overview

BASEL's design follows existing buffering architectures by considering only two types of objects: *ports*, and *queues* assigned to ports; in the buffered crossbar architecture [22, 23], cross-points can also be represented as ports. An *admission control policy* for a queue determines which packets are admitted or dropped [12, 14, 35]. A *scheduling policy* for a port selects a queue whose *head-of-line* (HOL) packet will be processed next [9, 31]; in each queue, the HOL packet is defined by a *processing policy*. Shared memory switches with several queues sharing the same *buffer space* [4, 10, 11] and architectures with synchronous management policies [24, 26] are out of scope of this paper.

In summary, to define a buffering architecture and its management policy one needs to create instances of ports, queues, and buffers, and specify relations among them; admission control, processing, and scheduling policies attached to the corresponding instances. These constructs suffice to achieve EXPRESSIVITY (cf. Section 1).

Fortunately, buffer management policies are generally concerned with boundary conditions (e.g., for admission a packet with *smallest* value can be dropped; to implement FIFO processing order, a packet with *smallest* arrival time is chosen next). Hence, *priority queues* arise as a natural choice for implementing actions related to the user-defined priorities. The priority criteria does not change at runtime (e.g., a queue's order can not be changed from FIFO to LIFO). We believe that this is a reasonable compromise to achieve conciseness for the policies without compromising expressiveness and performance. Each admission, processing, and scheduling policy in BASEL thus maintains its priority queue whose behavior is defined by a *comparator* – a Boolean function comparing two objects of same type via arithmetic/Boolean operators and accessing packet and object attributes.

2.2 BASEL API

In the following we present how BASEL's abstractions achieve SIMPLICITY (cf. Sec. 1) by means of simple declarations of data

```
Queue {
 // user-specified at declaration
 size           // size in bytes          [r, cons]

  // primitive properties
 currSize       // current size            [r, dyn]
 getHOL()       // head-of-line pkt    [packet fun]

 // admission -  user-specified at declaration
 congestion()   // congestion predicate [bool fun]
 admPrio(p1,p2) // pushOut prio comp.    [bool fun]
 postAdmAct()   // {MARK,NOTIFY,..}     [action fun]
 weightAdm      // priority for adm.       [rw, dyn]

 // processing -  user-specified at declaration
 procPrio(p1,p2)// process. prio comp.   [bool fun]

 // scheduling - user-specified at declaration
 weightSched    // prio. for scheduling   [rw, dyn]
}
```

Listing 1: BASEL's queue primitive.

```
Port {
 // primitive properties
 getBestQueue() // on weightSched    [queue fun]
 getCurrQueue() // scheduled one     [queue fun]

 // scheduling user-specified at declaration
 schedPrio(q1,q2)// compare q-s       [bool fun]
 postSchedAct() //{MARK,NOTIFY,..} [action fun]
}
```

Listing 2: BASEL's port primitive.

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

Listing 3: BASEL's packet primitive.

structures. For each entity, we define its properties, some of which are primitives of the domain (e.g., packet size), and others which have to be set by the programmer[1]. For functions we provide the return type (e.g., **`bool fun`**).

2.2.1 Queues

List. 1 summarizes the API to declare queues. The standard property `size` is defined by the user at declaration time. The `currSize` property changes dynamically as the queue changes its size. Abstractly, a queue contains packets ordered according to user-defined priorities for admission control and processing. In BASEL, we consider two user-defined priorities at the queue level:

(a) `procPrio(p1,p2)` is 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, so the only way to access a queue ordered by `procPrio` is through the `getHOL()` primitive which returns the HOL (i.e., highest processing priority as defined by `procPrio`) packet in the queue. E.g., the user can set

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

to encode FIFO processing. Hence, each call to `getHOL()` returns the packet with the oldest arrival time.

(b) `admPrio(p1,p2)` is also a packet comparator used in case of congestion to choose the packets that should be dropped from the queue. We could have simply chosen to use the least valuable packets according to `procPrio` for drops, but we will see in Sec. 3 that separate priorities for admission and processing gives more flexibility and improves performance.

The user-defined predicate `congestion()` indicates when a queue is virtually *congested*. Usually, `congestion()` is a set of different buffer occupancies and drop probabilities [14]. A capability to *push out* already admitted packets is supported in BASEL. To avoid different implementations for the push-out and non-push-out cases, an admission control policy always virtually admits an incoming packet. In the event of a virtual congestion, admission control drops the least valuable packets until congestion is lifted.

[1]For each property we indicate in comments whether it is **`r`** read-only or **`rw`** writable, and **`cons`** if it's value is fix at runtime, or **`dyn`** if it's value can change.

The optional function `postAdmAct()` returns an action applied after admission and can update `weightAdm` (if necessary). Function `postAdmAct()` can also be used to implement *explicit congestion notifications* [6] or *backpressure*; `postAdmAct()` can return actions such as **`MARK`** or **`NOTIFY`**. For cases when bandwidth is allocated not only with respect to packet attributes, queues maintain a `weightSched` variable that can be updated dynamically after each scheduling operation. With `weightSched` one can for example define static bandwidth allocation among queues of the same port during scheduling decisions; `weightSched` can be updated in the `postSchedAct()` function defined at the port level.

2.2.2 Ports

The interface provided for ports is presented in List. 2. A port manages a set of queues assigned at its declaration.[2] A user-defined scheduling property `schedPrio(q1,q2)` (queue comparator) defines which HOL packet is scheduled next (this packet is accessed through function `getBestQueue()`). For example, a priority based on packet values which implements several levels of strict priorities is declared simply as follows:

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

Finally, `postSchedAct()` is similar to the `postAdmAct()` function of queues which can be used to define new services.

2.2.3 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. Every incoming packet is prepended with three mandatory parameters — an *arrival* time, a packet *size* in bytes, and a destination *queue* — and three optional parameters — an intrinsic *value* (whose meaning is application-specific), the *processing* requirement in virtual cycles, and *slack* (maximal offset in time from *arrival* to transmission). We assume that these properties are set by an external *classification unit* (e.g., OpenFlow [32], if a virtual switch is defined with the finest possible resolution), except for *arrival* (set by BASEL when a packet is received) and *size*.

[2]We leave the `new` operator used to create network objects in BASEL implicit; its usage will be clear from the examples in Sec. 3.

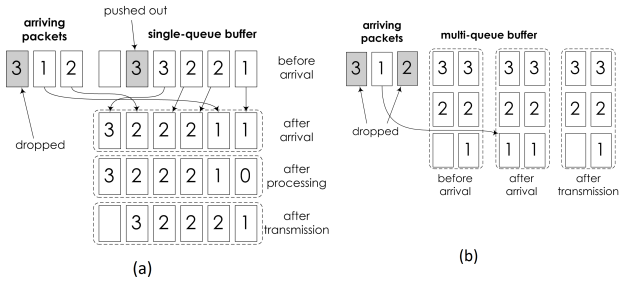

Figure 1: Left: single priority queue with buffer $B = 6$; right: multi-queued switch with three queues ($k = 3$) and buffer $B = 2$ each. Dashed lines enclose queues.

List. 3 depicts the `Packet` data structure. Intrinsic *value* and *processing* requirements can be useful to define prioritization levels [8, 23]. *Slack* is a time bound which 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 [25]. Sec. 3 shows specific examples exploiting some of these characteristics.

We postulate that all decisions of buffer management policies (during admission, processing, or scheduling) are based only on the specified packet parameters and internal state variables of a buffering architecture (e.g., buffer occupancy).

3. BASEL AT WORK (EXAMPLES)

3.1 Performance Impact of Admission Control

Consider throughput maximization in a single queue buffering architecture (buffer 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 BASEL 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. 4 shows the comparators and congestion conditions used in the following examples.

```
// 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 condition for all policies considered
// satisfied when occupancy exceeds queue size.
defCongestion() = lambda q, (q.currSize >= q.size)
```

Listing 4: Example priorities and congestion conditions

List. 5 shows the full specification of a single queue buffering architecture and its optimal throughput policy.

Table 1 lists implementations for `admPrio` and `procPrio` in this architecture and analytic competitiveness results for various online policies versus the optimal offline OPT algorithm [21, 27, 29, 30]. 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

```
// Specification of the buffering architecture
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 5: Single queue: optimal buffer managment policy for throughtput optimization.

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

Table 1: Sample BASEL policies for single queue architecture; k is the maximal processing requirement, OPT/ALG is the competitive ratio between the throughput of optimal offline OPT and online algorithm ALG.

processes them in `fifo()` order; it is $O(k)$-competitive for maximum processing requirement k. In BASEL, this algorithm looks as follows:

```
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 [21]. Since here a port manages only one queue, a *scheduling policy* is just an implicit call to `getHOL()`.

3.2 Performance Impact of Scheduling

One alternative architecture for packets with heterogeneous processing requirements is to allocate queues for packets with the same processing requirements (see Figure 1(b)). The following code creates this buffering architecture in BASEL, where `k` queues share an equal portion of memory `B`.

```
q1=Queue(B);...qk=Queue(B);
out=Port(q1,..,qk);
```

In this architecture, there is no need for advanced processing and admission orders since only packets with the same processing requirement are admitted in the same queue. The following BASEL code instantiates `admPrio`, `procPrio`, and `congestion` in the k created queues.

```
q1.admPrio=fifo; ...; qk.admPrio=fifo;
q1.procPrio=fifo; ...; qk.procPrio=fifo;
q1.congestion=defCongestion(q1); ...;
qk.congestion=defCongestion(qk);
```

This change of buffering architecture is not for free since 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 is most relevant for throughput optimization: buffer occupancy, required processing, or a combination. BASEL code in List. 6 presents six different scheduling priorities and `postSchedAct` actions in the cases when this action is used.

Table 2 summarizes various online scheduling policies as shown in [28]. Observe that buffer occupancy is not a good characteristic for throughput maximization: `lqf()` and `sqf()` have bad

```
// 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,
         (port.getCurrQueue().weightSched += k);
// PRR: Round-Robin with per packet resolution
prr(q1,q2)  = (q1.weightSched < q2.weightSched);
prrPostSchedAct() = lambda port,
  (let q = port.getCurrQueue() in
    if (q.getHOL().processing == 0)
       q.weightSched += k*k));
```

Listing 6: BASEL example of scheduling priorities and `postSchedAct` actions for multiple separated queues.

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 2: Examples of policies in BASEL for multiple queues architecture. k is the maximal processing requirements, B is a buffer size of a single queue. OPT/ALG is the throughtput of an optimal offline OPT algorithm vs. online algorithm ALG.

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 [28]. The other two policies that implement fairness with per-cycle or per-packet resolution (CRR and PRR respectively) have relatively weak performance; this demonstrates the fundamental tradeoff between fairness and throughput. The following code snippet in BASEL, for instance, corresponds to the CRR policy:

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

Currently, the best tools available to evaluate performance of buffering architectures are discrete simulators such as NS-2 [3] or OMNet++ [1] that can use traffic traces and/or various traffic distributions to analyze performance of buffer management policies in a high level language. Due to its simplicity, BASEL can be used as a discrete simulator whose configuration is limited to several user-defined expressions. For instance, Fig.s 2 and 3 show the impact of admission, processing, and scheduling policies on throughput optimization for a single queue and multiple queues buffering architectures with packets of heterogeneous processing requirements; in these examples, traffic was generated with an ON-OFF Markov modulated Poisson process (MMPP) with Poisson arrival processes with intensity λ, and required processing chosen uniformly at random from $1..k$. But even if we know how to represent arrivals and analyze them, the applicability of these results will be limited to specific settings. Hence, BASEL is being developed for deployment on real systems.

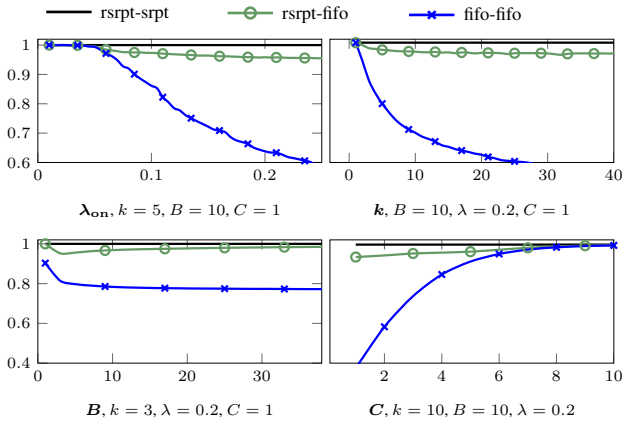

Figure 2: Optimal vs three online algorithms for a single queue architecture with heterogeneous processing; y-axis, competitive ratio; x-axis, top to bottom, left to right: λ; max required processing k; buffer size B; speedup C.

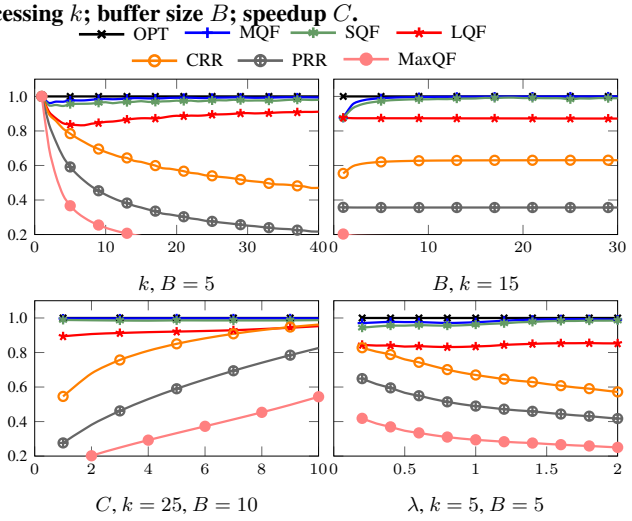

Figure 3: Online vs optimal algorithms for multiple queues with heterogeneous processing; y-axis, competitive ratios; x-axis, top to bottom, left to right: max required processing k, buffer size B, speedup C, intensity λ.

4. FEASIBILITY OF BASEL

A fundamental building block in BASEL is the *priority queue* data structure where the order of elements is based on user-defined priority. The implementation keeps a single copy of each packet and uses pointers to encode priorities (see Fig. 4). So BASEL implementation is reduced to efficient implementation of priority queues [19, 34] (cf. PERFORMANCE, Sec. 1), making BASEL attractive for hardware implementation.

4.1 BASEL Implementation in Open vSwitch

Open vSwitch (OVS) implements the control plane in user space and the data plane in the kernel [2, 37]. Since OVS exploits Linux TC (Traffic Control) kernel modules via the `netdev-linux` library to manipulate queuing and scheduling disciplines (`qdisc`),

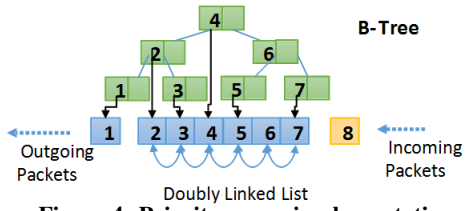

Figure 4: Priority queue implementation.

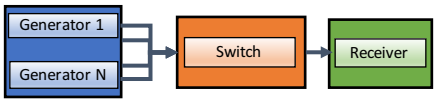

Figure 5: Testbed: 3-node topology

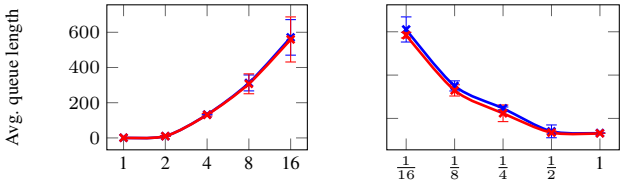

Figure 6: *Left:* average queue length as a function of number of clients generating UDP traffic with default MTU size. *Right:* fraction of default MTU size; blue: FIFO with prioritization; red: regular FIFO.

we have added configuration options to TC to express BASEL's admission, processing, and scheduling policies. Similar extensions are being added on the data plane via Linux kernel TC loadable kernel modules.

4.2 Performance Impact of Priority Queue

We have extended Linux's default qdisc[3] (i.e., pfifo_fast) to support packet prioritization based on arrival time. Instead of modifying the underlying default packet queue (a doubly linked list), we use an existing B-Tree implementation on top of a default FIFO queue to manage packet prioritization while preserving backward compatibility to existing qdisc solutions. As shown on Fig. 4, we add a reference to the enqueuing packets to the B-Tree and the highest priority packet (i.e., the earliest arrival time) is dequeued first. We remark that FIFO does not need to utilize a B-Tree in general; we use it as a baseline to explore the performance overhead of a generic implementation of prioritization.

In our testbed we set a 3-node line topology to measure the performance overhead of our packet prioritization logic. Fig. 5 shows that the middle node runs 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) in a receiver node on the third for two qdiscs: default FIFO and extended FIFO with prioritization, reporting the average value of 50 runs with 95% confidence interval. Fig. 6(left) shows the average queue lengths for the two qdiscs; in both cases, average queue length increases with the number of UDP clients. In FIFO with 16 clients, the most congested case, regular FIFO has average queue length 559.333 vs. 571 for FIFO with prioritization, only a 2% degradation. We also varied MTU sizes in the same 3-node line topology testbed with 4 parallel UDP generators, which is a good enough case to observe queue build-ups but not dropping packets in the pass-though switch.

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

We measured average queue lengths of the two qdiscs by varying MTU sizes from $\frac{1}{16}$ of the default MTU size to its default size (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. Hence, we conclude that packet prioritization on top of FIFO incurs negligible performance overhead.

5. RELATED WORK

The active networks [42] approach to programmable networks is to execute code contained within packets on the switches. However, we argue that running arbitrary code can hamper switch performance. Frenetic [15], Pyretic [33], among others, have proposed abstractions to express management policies in packet networks. These approaches focus on service abstractions based on flexible *classifiers*, and do not try to manage buffering architectures. Other systems [13,41] allow for setting a *predefined set* of parameters for buffer management, which intrinsically limits expressivity. Another line of research abstracts the representation of the southbound API (e.g., OpenFlow) in the data plane [7,40], while languages such as P4 [7] are very successful in representing packet classifiers, they are less suitable to express buffer management policies. The closest work to BASEL is [39] which introduces a set of primitives to specify only admission control policies for a single queue buffering architecture. On the other hand BASEL considers a composition of admission control, processing, and scheduling policies to optimize desired objectives on user-defined buffering architectures. Various frameworks have proposed mechanisms for specifying desired policies in packet networks such as bandwidth allocations [5,38].

6. CONCLUSION

We propose a concise yet expressive language to define buffer management policies at runtime. The proposed language can define buffering architectures and their management policies with any resolution from a single network element to a virtual switch that can represent a part of the network. We believe that BASEL can enable and accelerate innovation in the domain of buffering architectures and management, similar to programming abstractions that exploit OpenFlow for services with sophisticated classification modules. The conciseness of BASEL and ability to implement priority queue data structures at line-rate, make BASEL attractive for hardware implementations.

Acknowledgments

The work of S. Nikolenko was supported by by the Government of the Russian Federation grant 14.Z50.31.0030 and by the President Grant for Young Ph.D. Researchers MK-7287.2016.1. P. Eugster was partly funded by ERC grant "Lightweight Verification of Distributed Software" and German Research Foundation grant "Multi-Mechanism Adaptation for Future Internet".

7. REFERENCES

[1] OMNeT++. http://www.omnetpp.org/.
[2] Open vSwitch. http://www.openvswitch.org.
[3] This is the ns-2 wiki. http://nsnam.isi.edu/nsnam/index.php/Main_Page.
[4] W. Aiello, A. Kesselman, and Y. Mansour. Competitive buffer management for shared-memory switches. *ACM Trans. on Algorithms*, 5(1), 2008.

[5] H. Ballani, P. Costa, T. Karagiannis, and A. I. T. Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, pages 242–253, 2011.

[6] S. Bauer, R. Beverly, and A. Berger. Measuring the state of ECN readiness in servers, clients, and routers. In *IMC*, pages 171–180, 2011.

[7] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *CCR*, 44(3):87–95, 2014.

[8] P. Chuprikov, S. Nikolenko, and K. Kogan. Priority queueing with multiple packet characteristics. In *INFOCOM*, pages 1418–1426, 2015.

[9] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, pages 1–12, 1989.

[10] P. Eugster, A. Kesselman, K. Kogan, S. Nikolenko, and A. Sirotkin. Essential traffic parameters for shared memory switch performance. In *SIROCCO*, pages 61–75, 2015.

[11] P. Eugster, K. Kogan, S. Nikolenko, and A. Sirotkin. Shared memory buffer management for heterogeneous packet processing. In *ICDCS*, pages 471–480, 2014.

[12] W. Feng, K. G. Shin, D. D. Kandlur, and D. Saha. The BLUE active queue management algorithms. *IEEE/ACM Trans. Netw.*, 10(4):513–528, 2002.

[13] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: an API for application control of sdns. In *SIGCOMM*, pages 327–338, 2013.

[14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

[15] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *ICFP*, pages 279–291, 2011.

[16] J. Gettys. Low latency requires smart queuing:traditional AQM is not enough!, 2013. http://www.internetsociety.org/sites/default/files/pdf/accepted/29_bis_ISOC_Workshop_2.pdf.

[17] M. Goldwasser. A survey of buffer management policies for packet switches. *SIGACT News*, 41(1):100–128, 2010.

[18] C. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, pages 15–26, 2013.

[19] 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.

[20] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: experience with a globally-deployed software defined wan. In *SIGCOMM*, pages 3–14, 2013.

[21] I. Keslassy, K. Kogan, G. Scalosub, and M. Segal. Providing performance guarantees in multipass network processors. *IEEE/ACM Trans. Netw.*, 20(6):1895–1909, 2012.

[22] A. Kesselman, K. Kogan, and M. Segal. Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing. *Distributed Computing*, 23(3):163–175, 2010.

[23] A. Kesselman, K. Kogan, and M. Segal. Best effort and priority queuing policies for buffered crossbar switches. *Chicago J. Theor. Comput. Sci.*, 2012, 2012.

[24] A. Kesselman, K. Kogan, and M. Segal. Improved competitive performance bounds for cioq switches. *Algorithmica*, 63(1-2):411–424, 2012.

[25] A. Kesselman, Z. Lotker, Y. Mansour, B. Patt-Shamir, B. Schieber, and M. Sviridenko. Buffer overflow management in QoS switches. In *STOC*, pages 520–529, 2001.

[26] A. Kesselman and A. Rosén. Scheduling policies for CIOQ switches. *J. Algorithms*, 60(1):60–83, 2006.

[27] K. Kogan, A. López-Ortiz, S. Nikolenko, G. Scalosub, and M. Segal. Balancing work and size with bounded buffers. In *COMSNETS*, pages 1–8, 2014.

[28] K. Kogan, A. López-Ortiz, S. Nikolenko, and A. Sirotkin. Multi-queued network processors for packets with heterogeneous processing requirements. In *COMSNETS*, pages 1–10, 2013.

[29] K. Kogan, A. López-Ortiz, S. Nikolenko, A. Sirotkin, and D. Tugaryov. FIFO queueing policies for packets with heterogeneous processing. In *MedAlg*, pages 248–260, 2012.

[30] K. Kogan, A. López-Ortiz, S. I. Nikolenko, and A. Sirotkin. A taxonomy of semi-fifo policies. In *IPCCC*, pages 295–304, 2012.

[31] P. McKenney. Stochastic fairness queueing. In *INFOCOM*, pages 733–740, 1990.

[32] N. McKeown, G. Parulkar, S. Shenker, T. Anderson, L. Peterson, J. Turner, H. Balakrishnan, and J. Rexford. OpenFlow switch specification, 2011. http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[33] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software defined networks. In *NSDI*, pages 1–13, 2013.

[34] A. Morton, J. Liu, and I. Song. Efficient priority-queue data structure for hardware implementation. In *FPL*, pages 476–479, 2007.

[35] K. M. Nichols and V. Jacobson. Controlling queue delay. *Commun. ACM*, 55(7):42–50, 2012.

[36] S. I. Nikolenko and K. Kogan. Single and multiple buffer processing. In *Encyclopedia of Algorithms*. Springer, 2015.

[37] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *USENIX NSDI*, pages 117–130, 2015.

[38] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: sharing the network in cloud computing. In *SIGCOMM*, pages 187–198, 2012.

[39] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No silver bullet: extending SDN to the data plane. In *HotNets*, pages 19:1–19:7, 2013.

[40] H. Song. Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane. In *HotSDN*, pages 127–132, 2013.

[41] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. D. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *CoNEXT*, pages 213–226, 2014.

[42] D. L. Tennenhouse and D. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–17, 1996.