A Fast, Single-Instruction–Multiple-Data, Scalable Priority Queue

Imad Benacer, *Student Member, IEEE*, François-Raymond Boyer, and Yvon Savaria, *Fellow, IEEE*

***Abstract*— In this paper, we address a key challenge in designing flow-based traffic managers (TMs) for next-generation networks. One key functionality of a TM is to schedule the departure of packets on egress ports. This scheduling ensures that packets are sent in a way that meets the allowed bandwidth quotas for each flow. A TM handles policing, shaping, scheduling, and queuing. The latter is a core function in traffic management and is a bottleneck in the context of high-speed network devices. Aiming at high throughput and low latency, we propose a single-instruction–multiple-data (SIMD) hardware priority queue (PQ) to sort out packets in real time, supporting independently the three basic operations of enqueuing, dequeuing, and replacing in a single clock cycle. A proof of validity of the proposed hardware PQ data structure is presented. The implemented PQ architecture is coded in C++. Vivado high-level synthesis is used to generate synthesizable register transfer logic from the C++ model. This implementation on a ZC706 field-programmable gate array (FPGA) shows the scalability of the proposed solution for various queue depths with almost constant performance. It offers a 10× throughput improvement when compared to prior works, and it supports links operating at 100 Gb/s.**

***Index Terms*— Field-programmable gate array (FPGA), flow-based networking, high-level synthesis, priority queue (PQ), traffic manager (TM).**

I. Introduction

With the increasing number of Internet and mobile service subscribers, demand for high-speed data rates and advanced applications such as video sharing and streaming is growing fast. There is an ongoing process to define the next-generation communication infrastructure (5G) to cope with this demand. Yet, it is obvious that very low-latency packet switching and routing will be a major challenge to support life critical systems and real-time applications in the 5G context [1].

In network processing units (NPUs), packets are normally processed at wire speed through different modules. For example, a traffic manager (TM) provides queuing and scheduling functionalities [2], [3]; this is quite demanding because packet scheduling priorities are implicit and depend on several factors (protocols, traffic, congestion, etc.). One of the feasible solutions is to tag related packets with flow numbers [2] as soon as they enter the network. This helps allocating bandwidth and simplifies scheduling by alleviating the processing of individual packets in terms of flows or simply groups of packets.

With the current thrust toward software-defined networking [4], it becomes natural to associate each group of packets to a flow. For example, in cellular networks, bandwidth is assigned to subscribers, so each packet is already part of a flow with some bandwidth assigned to it. Thus, this flow tagging could become part of the context of the next-generation networking equipment.

Real-time applications, such as video streaming, require quality-of-service (QoS) guarantees such as average throughput, end-to-end delay, and jitter. To provide QoS guarantees, network resource prioritization matching requirements must be achieved by assigning priorities to packets according to the corresponding incoming flow information, which can represent specific types of applications, services, etc. To implement this priority-based scheduling, priority queues (PQs) implemented in hardware have been used to maintain real-time sorting of queue elements at link speeds. Hence, a fast hardware priority queue is crucial in high-speed networking devices (more details are given in Section II).

The PQs have been used for applications such as task scheduling [5], real-time sorting [6], and event simulation [7], [8]. A PQ is an abstract data structure that allows insertion of new items and extraction in priority order. In the literature, different types of PQs have been proposed. Reported solutions span between the following: calendar queues [8], binary trees [9], shift registers [9]–[11], systolic arrays [9], [12], register-based arrays [13], and binary heaps [13]–[17]. Existing solutions can be partitioned in two classes: PQs with fixed time operations or processing time that do not depend on the queue depth (number of nodes) and those with variable processing time.

This paper presents the following contributions.

1) A fast register-based single-instruction–multiple-data (SIMD) PQ architecture, supporting the three basic operations of enqueuing, dequeuing, and replacing. Our novel approach has modified the sorting operation in a way to restore required invariants in a single clock

Manuscript received September 8, 2017; revised January 2, 2018 and March 30, 2018; accepted May 2, 2018. Date of publication June 7, 2018; date of current version September 25, 2018. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, in part by Prompt Québec, and in part by Ericsson Research Canada. *(Corresponding author: Imad Benacer.)*

I. Benacer and F.-R. Boyer are with the Department of Computer and Software Engineering, École Polytechnique de Montréal, Montréal, QC H3T 1J4, Canada (e-mail: imad.benacer@polymtl.ca; francois-raymond.boyer@polymtl.ca).

Y. Savaria is with the Department of Electrical Engineering, École Polytechnique de Montréal, Montréal, QC H3T 1J4, Canada (e-mail: yvon.savaria@polymtl.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2018.2838044

cycle for the entire queue. Also, we provide a detailed proof of the correctness of the PQ with respect to various desired properties. It will be shown that after placement and routing, the required clock period is almost constant regardless of the queue depth.

2) A configurable field-programmable gate array (FPGA)-based PQ implementation using high-level synthesis (HLS), entirely coded in C++ to facilitate implementation (by raising the level of abstraction) and provide more flexibility with faster design space exploration than other works seen in the literature, which use low-level coding in Verilog, VHDL, etc. [2], [3], [9], [12], [16].
3) A queuing management system capable of providing at least 103 Gb/s for 64-B sized packets (see Section VI). Also, a fixed, stable throughput, independent of the queue depth, is achieved as compared to other architectures [3], [9], [13]–[16].

The remainder of this paper is organized as follows. In Section II, we present a literature review of some existing traffic management and PQ implementations. In Section III, we describe the architecture of a generic TM with its underlying modules. In Section IV, we present our hardware PQ with the proof of its validity, and a tradeoff analysis in terms of space versus time complexities. In Section V, we present the HLS methodology, and the various explored directives/constraints to target desired resource usage and performance. In Section VI, hardware implementations of the proposed design with comparisons to existing works in the literature are discussed, and Section VII draws conclusions.

II. Related Work

In this section, we present different traffic management works and solutions seen in the literature, and then, we detail well-known priority queuing models and their expected performances.

A. Traffic Management Solutions

Traffic management implementation evolved from network processor units [31]–[33] to dedicate stand-alone solutions [34], [35], namely, as coprocessors. Current solutions use dedicated traffic management integrated within NPUs to speed-up traffic processing, with external memories for packet buffering and queuing purposes.

The available traffic management solutions in the literature are essentially commercial products [31]–[35], which are usually closed. Few works about traffic management were published by academia. Zhang *et al.* [3] proposed a complete TM architecture implemented in an FPGA platform. Zhang focused on the programmability and scalability of the architecture in relation to today's networking requirements. However, the queue manager (QM) slows down the entire system with at least 9 cycles per action to enqueue/dequeue a packet, while running at 133 MHz. This TM achieved around 8 Gb/s for minimum size packets of 64 B. Khan *et al.* [36] proposed a traffic management solution implemented with dedicated circuits, supporting 5 Gb/s with full-duplex capabilities. Khan showed all the design steps up to the physical realization of a TM circuit. As Khan opted for an application-specified integrated circuit (ASIC), the reported solution remains rigid and has limited applicability for supporting future networking needs, such as increasing traffic demand and link speeds.

TABLE I
Traffic Management Solutions

Company / Researcher	Platform	Throughput (*Gb/s*)
Zhang (2012) [3]	FPGA Virtex-5 (65 nm)	8
Agere (2002) [31]	ASIC	10
Broadcom (2012) [32]	ASIC	Up to 200
Mellanox (EZchip) (2015) [33]	ASIC	Up to 400
Xilinx (2006) [34]	FPGA Virtex-4 (90 nm)	NA
Altera (2005) [35]	FPGA Stratix II (90 nm)	10
Khan (2003) [36]	ASIC (150 nm)	5
Bay (2007) [37]	ASIC (110 nm)	Up to 50

Table I summarizes the TM solutions offered by commercial vendors and published by academia, along with the platform for which they were developed, and their maximum achievable throughput.

B. Priority Queues

Previous reported PQs can be classified as software- or hardware-based. Each class is further described in Sections II-B1 and II-B2.

1) Software Solutions: No software PQ implementation in the literature can handle large PQs, with latency and throughput compatible with the requirements of today's high-speed networking systems. Existing software implementations are mostly based on heaps [13], [16], [18], with their inherent O(log(n)) complexity per operation, or alternatively O(s), where n is the number of keys or packets in the queue nodes, and s is the size of the keys (priority).

Research turned to the design of efficient high rate, and large PQs obtained by the use of specialized hardware, such as ASICs and FPGAs. These PQs are reviewed in Section II-B2.

2) Hardware Priority Queues: Moon *et al.* [9] evaluated four scalable PQ architectures based on: (first-in first-outs) FIFOs, binary trees, shift registers, and systolic arrays. This author showed that the shift register architecture suffers from a heavy bus loading problem as each new element has to be broadcasted to all blocks. This increases the hardware complexity and decreases the operating speed of the queue. The systolic array overcomes the problem of bus loading at the cost of higher resource usage than the shift register, needed for comparator logic and storage requirements. Similar to our design, the systolic PQ does not fully sort in a single clock cycle, but still manages to enqueue and dequeue in a correct order and in constant time. On the other hand, the binary tree suffers from scaling problems including increased dequeue time and bus loading. The bus loading problem is due to the required distribution of new entries to each storage element in the storage block.

TABLE II
EXPECTED THEORETICAL PERFORMANCE FOR DIFFERENT PQS

Queue Type	Enqueue (expected, worst case)	Dequeue (expected, worst case)
Calendar queue	O(1), O(n)	O(1), O(n)
Skew heap	O(log(n)), O(n)	O(log(n)), O(n)
Implicit binary heap	O(1), O(log(n))	O(log(n))
Skip lists	O(log(n)), O(n)	O(1)
Splay tree	O(log(n)), O(n)	O(1)
Binary search tree	O(log(n)), O(n)	O(log(n)), O(n)
Systolic array	O(1)	O(1)
Shift register	O(1)	O(1)
Binary heap	O(1), O(log(n))	O(log(n))
PIFO queue	O(1)	O(1)

The FIFO PQ architecture described by Moon *et al.* [9] uses one FIFO buffer per priority level. All such buffers are linked to a priority encoder to select the highest priority buffer. Compared to other designs, this architecture suffers from scaling the number of priority levels instead of the number of elements, requiring more FIFOs and a larger priority encoder.

Brown [8] proposed the calendar queue, similar to the bucket sorting algorithm, operating on an array of lists that contains future events. It is designed to operate with O(1) average performance, but poorly performs with changing priority distribution. Also, extensive hardware support is required for larger priority values.

Sivaraman *et al.* [38] proposed the PIFO queue. A PIFO is a PQ that allows elements to be enqueued into an arbitrary position according to the elements ranks (the scheduling order or time), while dequeued elements are always from the head. The sorting algorithm, called flow scheduler, enables O(1) performance. However, extensive hardware support is required due to the full ordering of the queue elements, compared to the partial sort in Moon's work and in our design. PIFO manages to enqueue, dequeue, and replace in the correct order and in constant time.

Ioannou and Katevenis [16] proposed a pipelined heap manager architecture that exploits a classical heap data structure (binary tree), while Bhagwan and Lin [17] proposed a pipelined heap (p-heap) architecture (which is similar to a binary heap). These two implementations of pipelined PQs offer scalability and achieve high throughput, but at the cost of increased hardware complexity and performance degradation for larger priority values and queue depths.

Table II summarizes the expected theoretical results of some PQs already reported in the literature, with their expected and worst case behavior for enqueue and dequeue operations. More details are given in [9] and [19].

III. TRAFFIC MANAGER ARCHITECTURE

In this section, we present a generic TM architecture and its operations. Then, we describe the modules from which it is composed.

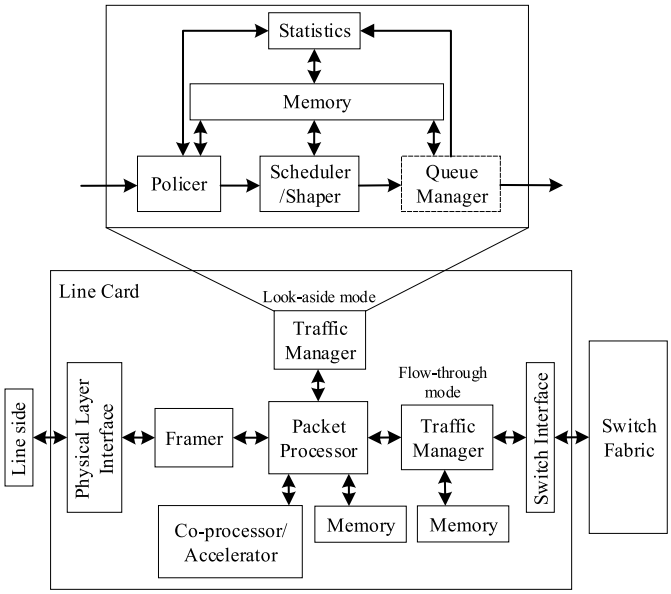

Fig. 1. Generic architecture around the TM on a line card. This paper focuses on the queue manager block.

A. Traffic Manager Overview

Traffic management allows bandwidth management, prioritizing, and regulating the outgoing traffic through the enforcement of service-level agreements (SLAs). An SLA defines the requirements that a network must meet for a specified customer or service, and it must be ensured, as a subscriber must get the level of service that was agreed upon with the service provider. The relevant criteria include, but are not limited to the performance metrics, such as guaranteed bandwidth, end-to-end delay, and jitter.

A generic TM in a line card (switch, router, etc.) is depicted in Fig. 1. In the flowthrough mode, the TM is in the data path. In the look-aside mode, the TM is outside the data path and communicates only with the packet processor, acting as a coprocessor. Generally, TMs reside on the line card next to the switch fabric interface, as they implement the output queuing necessary for the switch fabric or manage the packet buffers of the packet processor.

B. Structural Design

A packet processor is used to classify the data traffic to flows (flow tagging) prior its entry into the TM especially in the look-aside mode. A crude definition of a flow is a set of packets associated with a client of the infrastructure provider. These packets may be classified according to their header information. For example, packet classification is done by the packet processor using the five-tuple header information (source and destination IP, source and destination port, and protocol). The classified data traffic allows the TM to prioritize and decide how packets should be scheduled (i.e., when they should be sent to the switch fabric), how traffic should be shaped when sending packets onto the network, and which appropriate actions to take, for example, drop, retransmit, or forward.

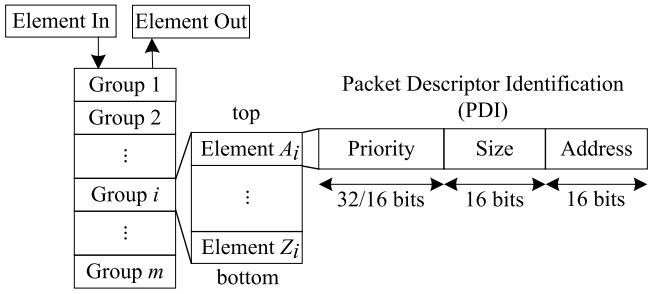

Fig. 2. SIMD register-array hardware PQ content.

1) Traffic Manager Operations: Traffic scheduling ensures, during times of congestion, that each port and each class of service (CoS) gets its fair share of bandwidth. The scheduler interacts with the QM block, notifying it of scheduling events. Packet congestion can cause severe network problems, including throughput degradation, increased delay, and high packet loss rates. Congestion management can improve network congestion by intelligently dropping packets.

The policer makes decisions on which packets to drop in order to prevent queue overflow and congestion. The shaper enforces packets to follow a specific network pattern by adding delays. The shaper imposes temporary delays to the outgoing traffic to ensure it fits a specific profile (link usage, bandwidth, etc.). During the TM operation, different statistics are being gathered for ingress and egress traffic in terms of received and transmitted packets, the number of discarded packets, etc. These data are stored for future analysis or system diagnosis.

The central piece of the TM is the QM. It maintains the packet priority sorted at link-speed using only the packet descriptor identification (PDI) [3]. The PDI enables packets to be located in the network through small metadata that can be processed in the data plane, while the entire packet is stored outside the TM in the packet buffer. This provides fast queue management with reduced buffering delays. The QM is responsible for packet's enqueue and dequeue or both at the same time. In this paper, we focus on the QM module in the TM of Fig. 1.

2) Packet Scheduling: The scheduler is responsible to tag, in the packet PDI, the new received packets, with a priority according to the scheduler policy. The PDI contains a priority field with 32 or 16 bits, the packet size in bytes expressed on 16 bits (to support any standard Internet packet size), the packet address location also expressed on 16 bits, and it may also contain other relevant attributes (see Fig. 2). This priority tagging may represent the expected departure time from the QM, as given by a scheduling algorithm [22], [23]. Also, this priority tagging may represent the CoS (voice, video, signaling, transactional data, network management, basic service, and low priority) for each packet, such that each classified packet corresponding to a flow is given a priority according to its respective traffic class. Packet classification and scheduling are not discussed further in this paper, as we focus on QM functionalities with the proposed SIMD PQ architecture.

3) Queue Manager: The QM presented in this paper performs enqueue operations of incoming PDIs. Also, the QM produces as outputs the PDIs of the packets that should be sent back to the network through either dequeue or replace operations. More details about the QM architecture are given in Section IV. For convenience, the term "packet PDI" is shortened to simply a "packet" in the next sections. It is worth noting that the QM's hardware PQ is composed of groups, each being connected with its adjacent groups, and each independently applying in parallel a common operation on its data. This architecture is register-based SIMD, with only local data interconnects, and a short broadcasted instruction. Thus, this would also qualify the proposed QM as a systolic architecture by some definitions in the literature. In the proposed QM, a fast hardware SIMD PQ is used to sort the packets from highest to lowest in priority, namely, in ascending order.

IV. Basic Operations of the SIMD Priority Queue

In general, PQs used in network devices have two basic operations: enqueue and dequeue. An enqueue inserts an element into the queue with its priority. A dequeue extracts the top or highest priority element, and removes it from the queue. In this paper, a dequeue–enqueue operation, or simply a replace operation, is considered as a third basic operation. Most works in the literature consider only the two first basic operations [3], [9], [10], [15], [21], but a few considered the third operation [13], [16], [17] achieving a higher throughput with better scalability.

In this paper, our proposed SIMD hardware PQ supports the following operations.

1) Enqueue (insert): A new element is inserted into the queue that is combined (partly sorted) with existing elements to restore the queue invariants (see Section IV-C).
2) Dequeue (extract min): The highest priority element is removed, and remaining elements are partly sorted to restore the queue invariants (see Section IV-C).
3) Replace (extract min with insertion): Similar to combined dequeue–enqueue operations, after which the number of elements inside the queue does not change. This operation is simultaneous for the insert and extract-min elements (see Section IV-B) while respecting the queue invariants (see Section IV-C).

The PQ behaves differently according to the operation to perform (the instruction). It is divided into m groups (see Fig. 2); a group contains N packets $A_g \ldots Z_g$, where g is the group number, A_g and Z_g represent the min and max elements, respectively, and a subset $\mathcal{S}$ containing the remaining elements in any order, namely, a group X_i contains N elements $\{A_i, \mathcal{S}_i, Z_i\}$ with $\mathcal{S}_i = \{X_i \setminus \{\min X_i, \max X_i\}\}$. The letters $A \ldots Z$ are used for generality regardless of the actual number of packets except in examples where N is known.

This architecture is based on the work we presented in [22], extended to add a third basic operation and generalizing to N packets in each group. Also, a proof of correct ordering of the queue elements is provided. For convenience, a queue supporting only the same operations as the previously reported

Element Out = A_1 (always, but not relevant on enqueue)

For all groups i ($i = 1, 2, \ldots m$) :

a) On enqueue operation:
group $i \leftarrow order\{ Z_{i-1}, A_i, \mathcal{S}_i \}$

b) On dequeue operation:
group $i \leftarrow order\{ \mathcal{S}_i, Z_i, A_{i+1} \}$

c) On replace operation:
group $i \leftarrow order\{\max\{Z_{i-1}, A_i\}, \mathcal{S}_i, \min\{Z_i, A_{i+1}\}\}$

Where $order\ X = \langle \min X, \mathcal{S}, \max X \rangle$,
$\mathcal{S} = X \backslash \{ \min X, \max X \}$,
Z_0 is the incoming packet (Element In),
A_{m+1} is the "none" packet equivalent to empty cell in the queue which must compare as greater (>) to any valid packet,
$\max\{Z_0, A_1\}$ is defined as Z_0 in c) since A_1 is dequeued.

Fig. 3. Hardware PQ algorithm.

architecture [22], without replace operation, is called an original PQ (OPQ).

A. Enqueue and Dequeue Operations

The algorithm of the OPQ is a combination of insert-and-sort or extract-and-sort for enqueue and dequeue operations, respectively. At every clock cycle, the queue accepts a new entry or returns the packet with the lowest priority value (the highest in priority). Packet movements obey the algorithm depicted in Fig. 3(a) and (b) for N packets in each group, see the Appendix for the notation.

In Fig. 3, Element Out is always connected to A_1 (top element) to reduce dequeue latency, but it is considered valid only on a dequeue or replace, not on an enqueue or no operation. The PQ just executes the same operation for all groups in parallel. All groups are ordered at the same time such that the OPQ enables independent enqueue and dequeue in constant time, regardless of the number of groups. Our implementation (see Section VI) does it in a single clock cycle for different queue depths and group sizes.

Note that the algorithm is well defined for any group size $N \geq 2$ and order is equivalent to fully sorting the elements of X only when $N \leq 3$. The unordered set has a single possible order in that case, it has 0 or 1 element for $N = 2$ and 3, respectively.

B. Replace Operation

An augmented PQ (APQ) is proposed to support the OPQ functionalities with the addition of a combined dequeue–enqueue operation, or simply replace, in the same clock cycle. In the case where both enqueue and dequeue must be performed on the OPQ with only two basic operations, the enqueue operation is prioritized over the dequeue, as to not lose the incoming packet. It should be noted that in case the OPQ is full, the lowest priority element is dropped as a consequence of the enqueuing. To overcome this issue of delaying the dequeue until no enqueue operation is activated in the same cycle, the third basic operation (replace) is proposed to deal with this case.

The algorithm of this APQ is a combination of insert, extract, and order for replace operation, in addition to the support of the enqueue and dequeue operations. The queue accepts a new entry and returns the packet with the highest priority at the same time. It does so correctly on a data set composed of a combination of the new entry combined with the current queue content. This can be done at every cycle (see Section VI). Packet movements obey the algorithm specified in Fig. 3(c). Note that for the last group m, the definition of A_{m+1} and the comparison operator implies that $\min\{Z_i, A_{i+1}\} = Z_i$. For enqueue operation, this last element is dropped when the PQ is full, but in a replace operation, no element is dropped as one element is dequeued and another is enqueued simultaneously.

Note that $\min\{A, B\}$ and $\max\{A, B\}$ functions must be defined such that one will return A and the other will return B, even when A and B are considered equal by the priority comparison operator.

An illustrative example of packet priorities movement in the OPQ and APQ is shown in Fig. 4 for few cycles, assuming the exemplary case of three packets in each group. Initially, the PQ contains empty cells. Empty cells priorities are represented by the maximum value of the priority. While time elapses cycle after cycle, the content of the PQ groups is displayed. It is of interest that the highest priority elements (smallest) remain close to the first groups, ready to exit, whereas the elements with lower priorities (largest) tend to migrate to the right of the queue. Meanwhile, it is worth noting that for the same example (see Fig. 4), the OPQ that does not support replace operation took more cycles and more storage elements as compared to the APQ.

C. Proof of Validity of the OPQ and APQ Behaviors

Queue invariants are provided to prove the correct functionality of the hardware PQ during enqueue, dequeue, and replace operations. The first two invariants will prove that the top element of the queue is always the highest priority one with no invariant ordering violation. The third invariant is provided to ensure that a drop may occur only in the situation where all the queue groups are full.

For all groups i with $i = 1, 2 \ldots m$ in the PQ, where each group contains N packets $\{A_i, \mathcal{S}_i, Z_i\}$, where $\mathcal{S}_i$ is a subset containing the remaining $N - 2$ elements of group i in any order.

1) Invariant 1: A_i and Z_i are, respectively, the highest and lowest priority elements in the group i. Note that in our case, the highest priority is the smallest value. That is

$$A_i = \min\{A_i, \mathcal{S}_i, Z_i\} \text{ and } Z_i = \max\{A_i, \mathcal{S}_i, Z_i\} \quad (1)$$

where $\{A_i, \mathcal{S}_i, Z_i\} = \{A_i, Z_i\} \cup \mathcal{S}_i$.

2) Invariant 2: Except Z_i, all elements in group i are of higher or equal priority than the first element in group $i + 1$. That is

$$\max\{A_i, \mathcal{S}_i\} \leq A_{i+1}. \quad (2)$$

Invariant 2 implies that $A_1 \leq A_2 \leq \ldots \leq A_m$, and by invariant 1, these A_i's are the minimum in their respective groups. Thus, A_1 is the smallest of all values.

	Cycle	In	Out	Group 1			Group 2			Group 3		
Insert	1	5										
Insert	2	7		5								
Insert	3	6		5	7							
Insert	4	2		5	6	7						
Extract	5		2	2	5	6	7					
Insert	6	1		5	6	7						
Insert	7	4		1	5	6	7					
Insert	8	0		1	4	5	6	7				
Insert	9	9		0	1	4	5	6	7			
Extract	10		0	0	1	9	4	5	6	7		
Extract	11		1	1	4	9	5	6	7			
Extract	12		4	4	5	9	6	7				
Extract	13		5	5	6	9	7					
Extract	14		6	6	7	9						
				7	9							

(a)

	Cycle	In	Out	Group 1			Group 2			Group 3		
Insert	1	5										
Insert	2	7		5								
Insert	3	6		5	7							
Insert	4	2		5	6	7						
Extract	5		2	2	5	6	7					
Replace	6	1	5	5	6	7						
Replace	7	4	1	1	6	7						
Replace	8	0	4	4	6	7						
Replace	9	9	0	0	6	7						
Extract	10		6	6	7	9						
				7	9							

(b)

Fig. 4. Example of packet priorities movement in (a) OPQ—packet priorities movement in the original SIMD and (b) APQ—packet priorities movement in the augmented SIMD PQ for few cycles.

So, the top element of the first group is the highest priority one in the PQ.

3) Invariant 3: A group i contains valid elements only if all the preceding groups are full.

We need to prove these invariants are preserved by the algorithm specified in Fig. 3. In the proof, we define that in the following.

a) $A_i \dots Z_i$ are the elements in group i before the operation, A_i being the first element of the vector, Z_i being the last.
b) G_i is the set of elements passed to the *order* function for group i in the algorithm (see Fig. 3).
c) $A'_i \dots Z'_i$ are the elements after the operation, thus the result of the *order* function.

We will prove that if $A_i \dots Z_i$ satisfy the invariants, $A'_i \dots Z'_i$ will also satisfy them. The initial state is an empty queue, where all elements in the queue compare equal to each other (represented by the maximum value of the priority), satisfying the invariants. We will prove by induction that the algorithm preserves those invariants for all operations.

1) Proof for Invariant 1: All three operations (enqueue, dequeue, and replace) do

$$\begin{aligned}\langle A'_i, \mathcal{S}'_i, Z'_i \rangle &= order\ G_i \\ &= \langle \min G_i, G_i \setminus \{\min G_i, \max G_i\}, \max G_i \rangle. \quad (3)\end{aligned}$$

Thus, $A'_i = \min G_i$, $Z'_i = \max G_i$ and the other elements are the remaining elements of G_i in any order represented with the subset $\mathcal{S}'_i$. After the operation, invariant 1 is thus satisfied regardless of whether invariants were satisfied or not before the operation.

2) Proof for Invariant 2: Supposing invariants are satisfied on $A_i \dots Z_i$, from (2) we have to verify whether $\max\{A'_i, \mathcal{S}'_i\} \leq A'_{i+1}$ for the three operations. By (3), A'_i and $\mathcal{S}'_i$ represent all elements of G_i except its max, thus $\max\{A'_i, \mathcal{S}'_i\}$ is the second largest element in G_i. We thus define 2nd max X as the second largest element in set X. Also by (3), $A'_{i+1} = \min G_{i+1}$, the above verification is equivalent to

$$\text{2nd } \max G_i \leq \min G_{i+1}. \quad (4)$$

In (4), as G_i is defined in terms of $A_i \dots Z_i$, not $A'_i \dots Z'_i$, by the induction hypothesis, we can use (1) and (2) on these variables. The following property will also be used:

$$\text{2nd } \max X \leq \max(X \setminus \text{any single element}). \quad (5)$$

In (5), if the single element removed was not the max of X, the max remains the same, and 2nd max $X \leq \max X$ by definition, and if it was the max of X, 2nd max $X = \max(X \setminus \max X)$ also by definition, proving (5).

a) Proof for the enqueue operation:

$$\begin{aligned}
&G_i = \{Z_{i-1}, A_i, \mathcal{S}_i\} \quad \text{by Fig. 3(a)} && .a\\
&\text{2nd max } G_i \leq \max\{A_i, \mathcal{S}_i\} \quad \text{by .a into (5)} && .b\\
&\quad " \quad \leq Z_i \quad \text{by (1) on .b} && .c\\
&\quad " \quad \leq A_{i+1} \quad \text{by (2) on .b} && .d\\
&\quad " \quad \leq \min\{A_{i+1}, \mathcal{S}_{i+1}\} \quad \text{by (1) on .d} && .e\\
&\quad " \quad \leq \min\{Z_i, A_{i+1}, \mathcal{S}_{i+1}\} \quad \text{by merging .c, .e} && .f\\
&\quad " \quad \leq \min\ G_{i+1} \quad \text{by .a on .f} && .g
\end{aligned}$$

By .g, we verified (4), thus invariant 2 is preserved.

b) Proof for the dequeue operation:

$$G_i = \{\mathcal{S}_i, Z_i, A_{i+1}\} \quad \text{by Fig. 3(b)} \quad .a$$
$$\text{2nd max}G_i \leq \max\{\mathcal{S}_i, A_{i+1}\} \quad \text{by .a into (5)} \quad .b$$
$$" \quad \leq A_{i+1} \quad \text{by (2) on .b} \quad .c$$
$$" \quad \leq \min\{\mathcal{S}_{i+1}, Z_{i+1}\} \quad \text{by (1) on .c} \quad .d$$
$$" \quad \leq A_{i+2} \quad \text{by (2) on .c} \quad .e$$
$$" \quad \leq \min\{\mathcal{S}_{i+1}, Z_{i+1}, A_{i+2}\} \quad \text{by merging .d, .e} \quad .f$$
$$" \quad \leq \min\ G_{i+1} \quad \text{by .a on .f} \quad .g$$

By .g, we verified (4), thus invariant 2 is preserved.

c) Proof for the replace operation:

$$G_i = \{\max\{Z_{i-1}, A_i\}, \mathcal{S}_i, \min\{Z_i, A_{i+1}\}\} \quad \text{by Fig. 3(c)} \quad .a$$

Left part of (4): 2nd max G_i

$$\text{2nd max}\ G_i \leq \max\{\mathcal{S}_i, \min\{Z_i, A_{i+1}\}\} \quad \text{by .a into (5)} \quad .b$$
$$\max\{\mathcal{S}_i\} \leq A_{i+1} \quad \text{by (2)} \quad .c$$
$$\min\{Z_i, A_{i+1}\} \leq A_{i+1} \quad \text{by definition of min} \quad .d$$
$$\max\{\mathcal{S}_i, \min\{Z_i, A_{i+1}\}\} \leq A_{i+1} \quad \text{by merging .c, .d} \quad .e$$
$$\text{2nd max}\ G_i \leq A_{i+1} \quad \text{by .e on .b} \quad .f$$

Right part of (4): min G_{i+1}

$$\min G_{i+1} = \min\{\max\{Z_i, A_{i+1}\}, \mathcal{S}_{i+1}, \min\{Z_{i+1}, A_{i+2}\}\} \quad \text{by .a} \quad .g$$
$$" \quad = \min\{\max\{Z_i, A_{i+1}\}, \mathcal{S}_{i+1}, Z_{i+1}, A_{i+2}\} \quad \text{by associativity on .g} \quad .h$$
$$A_{i+1} \leq \max\{Z_i, A_{i+1}\} \quad \text{by definition of max} \quad .i$$
$$" \quad \leq \min\{\mathcal{S}_{i+1}, Z_{i+1}\} \quad \text{by (1)} \quad .j$$
$$" \quad \leq A_{i+2} \quad \text{by (2)} \quad .k$$
$$" \quad \leq \min\ G_{i+1} \quad \text{by merging .i-.k into .h} \quad .m$$

By .f and .m, we verified (4), thus invariant 2 is preserved.

Note that the above proof does not use A_1 for valid values of $i(1 \ldots m)$, thus the special definition of $\max\{Z_0, A_1\}$ at the bottom of Fig. 3 has no influence on the proof.

We have thus proven that the algorithm preserves the invariants 1 and 2. For the algorithm to be proven correct, we also need to verify that the inserted elements are correctly conserved in the queue, not deleted nor duplicated. In Fig. 3, the *order* function clearly keeps all elements without duplication if the min and max removed to make $\mathcal{S}$ are the same as those placed in the first and last elements (remember that elements can have similar priorities, and thus compare as equal in the ordering, but having different associated metadata).

On dequeue [Fig. 3(b)]: A_1, the outgoing element, is correctly removed; A_{i+1} goes into group i, other elements stay in the same group; group m has an empty cell.

On replace [Fig. 3(c)]: A_1, the outgoing element, is correctly removed because of the special case at the bottom of Fig. 3; Z_i and A_{i+1} are in a min (in G_i) and a max (in G_{i+1}), and this will keep both element by the way we defined the min/max pair; for G_m, $\min\{Z_m, A_{m+1}\}$ correctly keeps Z_m if it is valid, by the definition of A_{m+1}.

On enqueue [Fig. 3(a)]: Z_0, the incoming element, enters in group 1; Z_i goes into group $i+1$, other elements stay in the same group; Z_m is dropped. It is important to show that Z_m will only contain a valid element when the queue is full, and thus, it would be required to drop an element during an enqueue.

3) Proof for Invariant 3: From invariant 1, where A_i and Z_i are, respectively, the smallest and largest in group i, and definition of "none" packet/element (bottom of Fig. 3) which states they compare as greater than any valid element, we deduce two remarks as follows.

Remark 1: A_i is valid if and only if group i contains at least one valid element.

Remark 2: Z_i is valid if and only if group i is full.

a) Proof for the enqueue operation: On enqueue [Fig. 3(a)]: $\langle A'_i, \mathcal{S}'_i, Z'_i\rangle = order\{Z_{i-1}, A_i, \mathcal{S}_i\}$. For group i to contain valid elements after the operation, two cases must be considered:

Case 1: Group i was not empty, thus A_i is valid by Remark 1, and A_i being still in group i after the operation, A'_i is also valid by Remark 1. By invariant 3 (induction hypothesis), groups $1 \ldots i-1$ were full ($A_1 \ldots Z_{i-1}$ are valid), and Z_0 is valid by definition of enqueue. Thus, groups preceding i are full ($A'_1 \ldots Z'_{i-1}$ being a reordering of valid $Z_0, A_1 \ldots \mathcal{S}_{i-1}$). Therefore, invariant 3 is preserved.

Case 2: Group i was empty, but Z_{i-1} is valid thus group $i-1$ was full by Remark 2. $A_1 \ldots Z_{i-2}$ are valid by invariant 3 and $A_{i-1} \ldots Z_{i-1}$ are valid as group $i-1$ was full, thus, as in previous case, it implies that invariant 3 is preserved.

b) Proof for the dequeue operation: On dequeue [Fig. 3(b)], A_{m+1} is the "none" element entering into the last group m per dequeue operation, with $\langle A'_i \ldots Z'_i\rangle = order\{\mathcal{S}_i, Z_i, A_{i+1}\}$.

For group i to contain valid elements after the operation, at least one element in the set passed to order must be valid. As A_{i+1} cannot be valid without $A_1 \ldots Z_i$ being valid (induction hypothesis), at least one element of $\{\mathcal{S}_i, Z_i\}$ is valid, and by remark 1, A_i is valid. By induction hypothesis, groups $1 \ldots i-1$ were full ($A_1 \ldots Z_{i-1}$ are valid). Thus, groups preceding i are full ($A'_1 \ldots Z'_{i-1}$ being a reordering of valid $\mathcal{S}_1 \ldots A_i$), and invariant 3 is preserved.

c) Proof for the replace operation: On replace [Fig. 3(c)], $\langle A'_i \ldots Z'_i\rangle = order\{\max\{Z_{i-1}, A_i\},\ \mathcal{S}_i,\ \min\{Z_i, A_{i+1}\}\}$. Two cases must be considered for group i before the operation as follows.

Case 1: Group i was not empty, by Remark 1, A_i is valid. By invariant 3, groups $1 \ldots i-1$ were full ($A_1 \ldots Z_{i-1}$ are valid), and Z_0 is valid by definition of replace. Thus, groups preceding group i are still full after the operation ($A'_1 \ldots Z'_{i-1}$ being a reordering of valid $Z_0, A_1 \ldots A_i$), and invariant 3 is preserved.

Case 2: Group i was empty. By Remark 1, A_i is empty, group i remains empty after the operation as the max $\{Z_{i-1}, A_i\}$ returns always a "none/empty" element (A_i). Thus, invariant 3 is preserved.

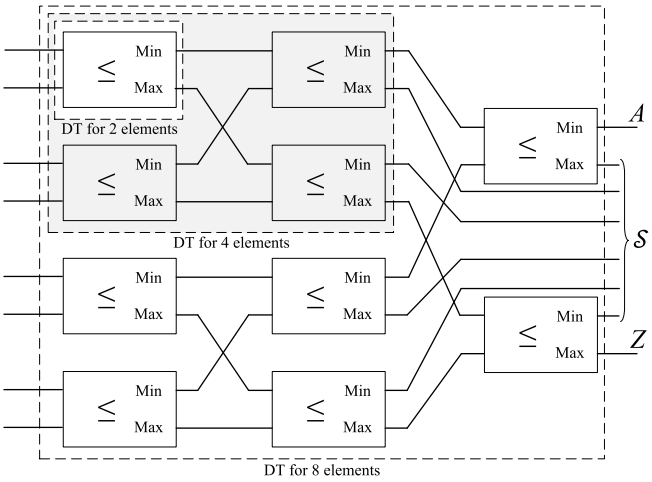

Fig. 5. Proposed *order* function architecture supporting two, four, and eight elements in each group.

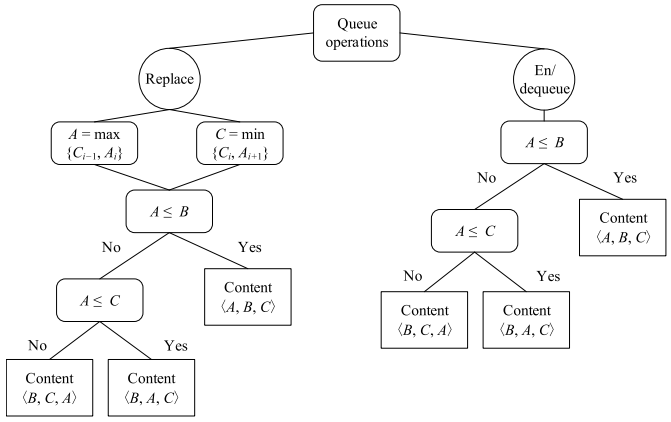

Fig. 6. Proposed DT giving the result of *order* $\langle A, B, C\rangle$, for three packets in each group of the PQ. Note that we used $\langle\rangle$ instead of $\{\}$ on the *order* function, because it can be optimized relying on known current ordering of elements in the group i. The element coming from another group is called A and the elements from current group are called B and C in priority order in the case of en/dequeue, whereas in replace only A and C are calculated from max/min, respectively, and B is in the current group.

If Z_m is valid, group m is full by Remark 2, and groups $1 \ldots m-1$ are full by invariant 3, meaning that the PQ is full. Therefore, an element (Z_m) will be dropped only in the case of an enqueue operation when the PQ is full.

We note that, from invariants 1 and 2, we have

$$A_1 \leq (\mathcal{S}_1) \leq A_2 \leq (\mathcal{S}_2) \leq A_3 \ldots, \text{ etc.}$$

The only unordered elements are the Z_i's, thus, in the worst case, Z_m is the mth lowest priority element. For a constant queue depth, increasing N reduces m, and the dropped element when enqueuing on a full queue will be of lower priority. Also, if the *order* function fully sorts the elements (which is always true for groups of size $N \leq 3$), the whole queue is sorted except the Z_i's.

D. Original and Augmented PQs Decision Tree

A decision tree (DT) is used to implement the ordering independently in each group of the PQ. The entire PQ's group elements are evaluated in parallel and at the same time according to the priority of each element. Increasing the number of elements in each group (the space complexity) will impact directly the DT and the overall queue performance (the time comlexity). Also, this impacts the quality of dismissed elements when the queue is full. This tradeoff is detailed in Section IV-D1.

1) Tradeoff (Time Versus Space): To choose the number of elements in each group, two things should be taken into consideration: performance and quality of dropped elements when the queue is full. The complexity of the proposed DT for sorting the elements, i.e., the *order* function depicted in Fig. 3, is O(log(N)). This DT belongs to the family of parallel networks for sorting [29], [30]. Fig. 5 depicts the *order* function scheme for $N = 2$, 4, and 8 elements, respectively.

The proposed DT can be further optimized for $N = 3$. In this special case, we take advantage of the present information on already ordered elements in the current group. This DT is depicted in Fig. 6 for the enqueue, dequeue, and replace operations in each group. On each side of the DT, a specific test is made. For example, the right side is dedicated for the group ordering when only en/dequeue operation is activated, and the left side is for the replace operation. The order is determined by comparing the priorities of the packets tag present in the different PQ groups using only two comparators. The overall architecture of the proposed hardware SIMD PQ is depicted in Fig. 7.

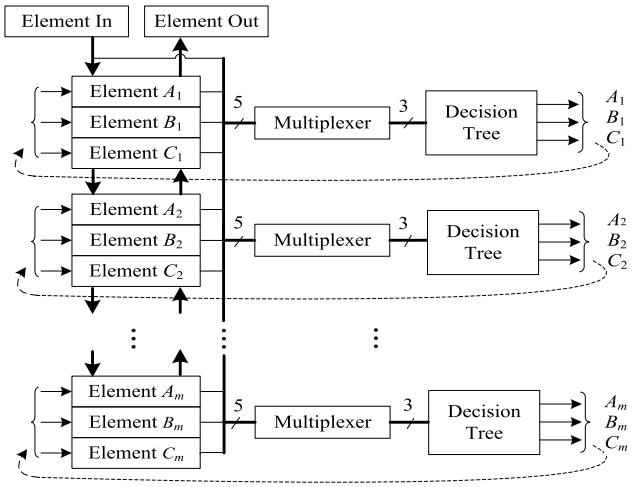

Fig. 7. SIMD PQ architecture for three packets in each group.

2) Quality of Dismissed Elements: To achieve good performance in terms of latency and number of cycles spent for operations (we target 1 cycle per operation), the proposed architecture is sacrificing two characteristics compared to previous reported approaches depending on the number of groups (m) and the size of the groups (N) for a constant queue depth as follows:

1) quality of dismissed elements if m is large (the number of elements N in a group is small);

2) resource usage if m is small (number of elements N in a group is large).

In the worst case, the dismissed element is the mth lower priority element in the queue (the bottom element in the last group m). The queue depth is $m \times N$. So, the quality of the dismissed element is calculated according to the following equation:

$$\text{Quality of dismiss} = \frac{\text{Number of groups}}{\text{Queue depth}} = \frac{m}{m \times N} = \frac{1}{N}. \tag{6}$$

For example, $N = 2$, the quality of dismiss is 50%, namely, the dropped element is within the 50% lower priority elements. However, for $N = 64$, this dropped element would be in the 1.56% lower priority elements. So, the higher is N, the best is for the quality, but the performance decreases in O(log(N)). More details about the experimental results are in Section VI.

V. HLS Design Methodology and Considerations

HLS allows raising the design abstraction level and flexibility of an implementation by automatically generating synthesizable register transfer logic (RTL) from C/C++ models. In addition, exploring the design space using the available directives and constraints allows the user to guide the HLS tool during synthesis. Also, HLS require less design effort, when performing a broad design space exploration as many derivative designs can be obtained with a small incremental effort. Once a suitable specified functionality has been derived, a designer can focus on the algorithmic design aspects rather than low-level details required when using a hardware description language (HDL).

The first step in any HLS design is the creation of high-level design description of the desired functionality. This description is typically subject to design iterations for refinement (code optimization and enhancement), verification and testing to eliminate bugs, errors, etc. Then, design implementation metrics should be defined such as the target resource usage, desired throughput, clock period, design latency, input–output requirements, etc., which are closely related to the design process, and that are in fact part of the design process. These metrics can be controlled through directives/constraints applied during HLS process. The HLS process can be described in two steps: 1) extraction of data and control paths from the high-level design files and 2) scheduling and binding of the RTL in the hardware, targeting a specific device library. In this paper, we performed all design experimentation with the Vivado HLS tool version 2016.2, while the design was coded in C++.

A. Design Space Exploration With HLS

Table III summarizes the results of design space exploration for the OPQ with two elements in each group, while the total specified queue capacity is 64 packets. The metrics used to measure performance in any HLS design are area, latency, and initiation interval (II). Partition directive is used to force the tool to use only logic resources with no BRAMs (on-chip block RAMs) even available in the FPGA. This reduces latency by cutting down the time to memory access. The unroll directive lead to parallelized design producing an output each clock cycle (II = 1), but at higher costs in terms of lookup tables (LUTs) compared to the previous directive. The pipeline directive gives similar results to unroll but it achieves the best clock period. Exploring combinations of the above cited directives with inline for *order* function gives similar results to unroll or pipeline, respectively. However, putting all four directives together in the right place in the code (pipelining the PQ design with II = 1, partition of the queue elements, unrolling the queue groups) gives the best design in terms of resource usage, and performance. These HLS results were generated for all queue configurations (OPQ and APQ) and for different queue depths ranging from 34 up to 1024, while the number of elements in each group varies from $N = 2, 3, \ldots 64$. More details on the experimental results of placement and routing in the FPGA are given in Section VI.

TABLE III
HLS Design Space Exploration Results of a 64-Element OPQ, With $N = 2$ and 64-bit PDI

Optimization	Resources			Performance		
	BRAM	*FFs*	*LUTs*	Latency Min / Max (Cycles)	II Min / Max (Cycles)	Clock (ns)
Default	6	368	1336	3 - 130	4 - 131	9.19
Partition (1)	0	12267	4992	2 - 95	3 - 96	6.76
Unroll (2)	0	4162	13089	0	1	5.68
Pipeline (3)	0	4098	13025			4.42
(1) + (2) + Inline (4)	0	4162	13089			5.98
(1) + (3)	0	4098	13025			4.42
(2) + (3) + (4)	0	4098	13025			4.42
(1) + (2) + (3) + (4)	0	4098	8621			4.42

B. Real Traffic Trace Analysis

In order to establish the parameters for the design (especially the queue depth), a detailed analysis was undertaken to find the number of packets that are seen in Internet traffic. This was done by examining real traffic traces with different rates, collected by CAIDA from OC-48 and OC-192 representing, respectively, 2.5- and 10-Gb/s links [20]. Table IV depicts the trace characteristics and the rate of packets seen with a monitoring window of 1 ms and 1 s, for 300- and 60-s traces duration for OC-48 and OC-192, respectively. A 1-ms monitoring window is sufficient to satisfy a requirement of high speed as packets are processed in ns time window. From Table IV, there are only ∼75 and 560 packets on average in OC-48 and OC-192 links, respectively, seen in a 1-ms time window. For a queue capacity of 1024 PDIs, it can support today's high-speed links requirement ranging from 2.5 up to 10 Gb/s.

VI. Experimental Results

In this section, we detail the hardware implementation of our proposed SIMD PQ, as well as its resource usage and

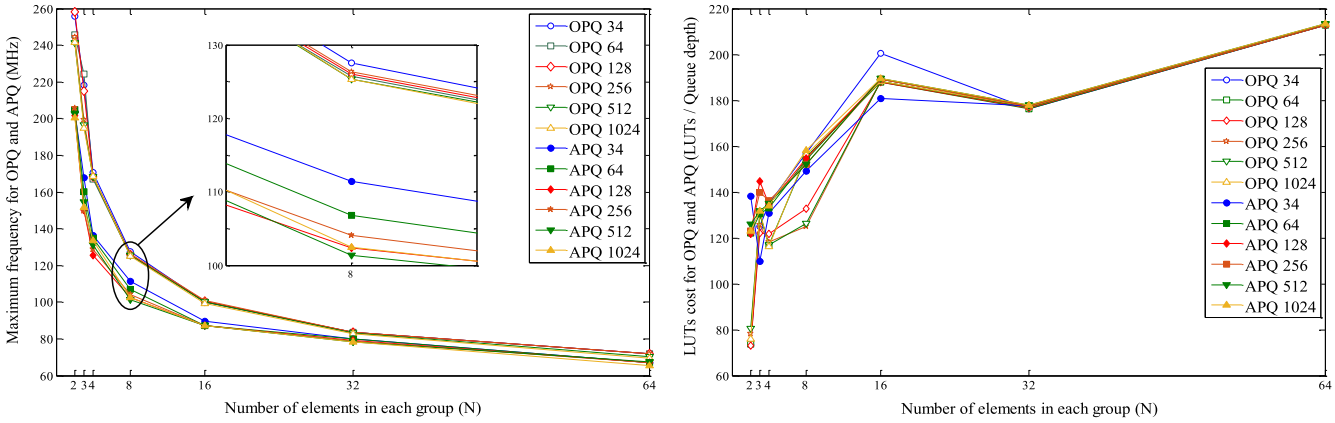

Fig. 8. Experimental results for different queue configurations (OPQ and APQ) with depths ranging from 34 up to 1 Ki. (a) Plots of the maximum frequency of operation for both queues. (b) LUTs cost per queue depth in terms of the number of elements in each group (N). For FFs cost, we obtained a constant 64-bit representing the size of one element (tag) in each group for both OPQ and APQ. These reported results are for 64-bit PDI, with 32-bit priority.

TABLE IV

TRAFFIC TRACE CHARACTERISTICS AND AVERAGE PACKETS SEEN IN 1-ms AND 1-s TIME INTERVALS FOR 300- AND 60-s DURATION FOR OC-48 AND OC-192 LINKS CAIDA [20] TRACES, RESPECTIVELY

Link	Date	Trace Characteristics		Number of Packets	
		Avg. Packet Len. (Byte)	*Avg. Rate (Mbit/s)*	1 ms	1 s
OC-48	24/04/2003	534	108	25	25248
	15/01/2003	685	324	59	59175
	14/08/2002	571	342	74	74989
OC-192	06/04/2016	833	3744	562	561752
	17/03/2016	601	2073	431	427131
	17/12/2015	662	2463	465	461312

TABLE V

RESOURCE UTILIZATION OF THE ORIGINAL AND AUGMENTED HARDWARE PQS, WITH 64-bit PDI

		Queue Depth	34	64	128	256	512	1 Ki
OPQ	$N=2$	LUTs	2494	4719	9362	20003	41257	77896
		FFs	2176	4096	8192	16384	32768	65536
		Utilization	1%	2%	4%	9%	19%	36%
		F_{max} (MHz)	256	246	258	244	241	242
	$N=3$	LUTs	4126	8363	15621	32127	66757	134726
		FFs	2112	4032	8064	16320	32640	65472
		Utilization	2%	4%	7%	15%	31%	62%
		F_{max} (MHz)	218	224	215	199	196	195
APQ	$N=2$	LUTs	4704	7818	15604	31495	64522	126129
		FFs	2176	4096	8192	16384	32768	65536
		Utilization	2%	4%	7%	14%	30%	58%
		F_{max} (MHz)	204	205	202	206	203	201
	$N=3$	LUTs	3630	8349	18523	35806	67514	135068
		FFs	2112	4032	8064	16320	32640	65472
		Utilization	2%	4%	8%	16%	31%	62%
		F_{max} (MHz)	168	160	150	150	154	152

achieved performance for different configurations (OPQ and APQ). Then, comparisons to existing works in the literature are made.

A. Placement and Routing Results

The proposed hardware SIMD PQ was implemented (placed and routed) on the Xilinx Zynq-7000 ZC706 evaluation board (based on the xc7z045ffg900-2 FPGA). The resource utilization of the implemented hardware PQ architecture for different queue depths are shown in Table V for OPQ and APQ, with two cases: $N = 2$ and 3 packets in each group. Each hardware PQ element has a 64-bit width, with 32 bit representing the priority, 16 bit for packet size, and 16 bit for the address (see Fig. 2). The queue depth is varied from 34 to 256, in order to allow comparing with [15]. Also, complexity of 512 and 1024 (1 Ki) deep PQs are summarized in Table V. Other experimental results for different configurations with N varying up to 64 with frequency of operation are shown in Fig. 8(a), and logic resource utilization in Fig. 8(b). Slices utilization for both OPQ and APQ configurations are depicted in Fig. 9(a) and (b), respectively.

When implementing the QM's hardware PQ, only flip-flops (FFs) and LUTs were used to obtain a fast, low latency architecture. The queue is able to take an input and provide an output in the same clock cycle, thus the proposed PQ implementation has a 0-cycle latency. This PQ is capable of performing all the three basic operations in a single clock cycle. Note that as mentioned earlier, the required clock period after placement and routing remains almost constant regardless of the queue depth, as depicted in Fig. 8(a) for different N ranging from 2 up to 64 elements in each group, for both OPQ and APQ, with queue depths varying from 34 up to 1 Ki.

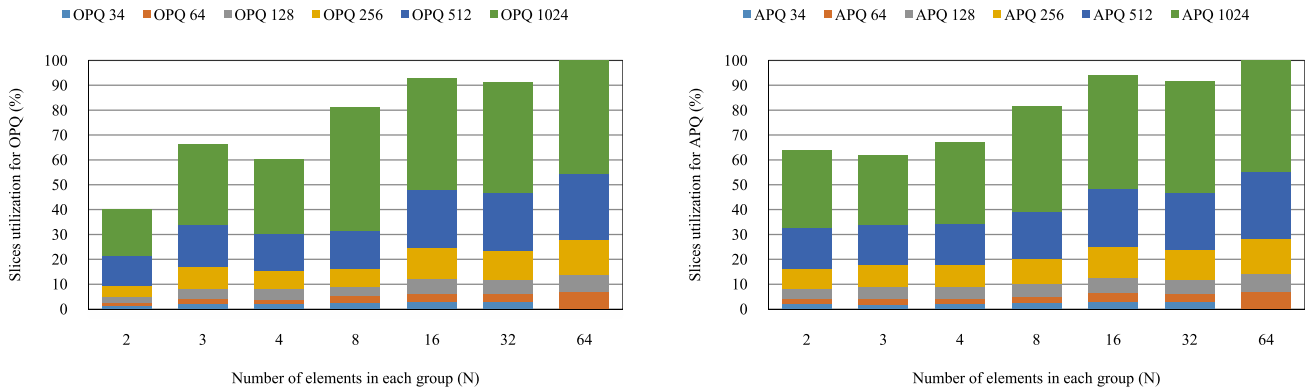

Fig. 9. Slices utilization results for different queue configurations. OPQ (left) and APQ (right) histogram with 64-bit PDI.

The operating frequency reported, after placement and routing (using Vivado 2016.2 with the *Explore* directive enabled), by the timing analysis tool degrades in O(log(N)) between a maximum 258 and a minimum 69 MHz for the OPQ with a group size of 2 up to 64. For the APQ, similar behavior is observed while the maximum frequency is 206 degrading to 65 MHz.

From Table V and Fig. 8(b), it is clearly seen that increasing the group size N, leads to an increase in LUTs consumption and not FFs (cost of FFs per elements remained constant and equal to 64-bit, i.e., reflecting the element size). This is due to the required logic in the DT for the additional packets in each group, leading to larger multiplexers and more levels of comparators. Also, the added replace operation to the hardware PQ does increase the LUTs consumption only for small $N < 16$. This LUT usage increase is related to the fact that the replace operation did not require architectural modification on the OPQ. Indeed, it only added a min and max calculation prior to the DT or the *order* function in each group. However, when N increases, the impact of the min/max decreases and the OPQ and APQ LUTs usage converges to nearly the same value when $N \geq 16$.

Fig. 9 summarizes the slice usage for the OPQ and APQ with different configurations (queue depth and number of elements N). It is of interest that both OPQ and APQ have similar slice usage for $N \geq 16$ (similar to the previous explanation of min/max influence needed in the replace operation). On the other hand, as the APQ is more complex than the OPQ, this min/max influence is mostly seen for $N < 16$. For $N = 3$, the slices usage for different queue capacities in OPQ can be lower compared to the APQ (as for 128 and 256), and it is always larger in the remaining queue capacities, this particular case is only observed for $N = 3$. A lower complexity APQ was only observed for this particular case.

The achieved frequency for the different hardware PQs are also reported as functions of queue depth in Table V and Fig. 8(a). The achieved throughput in the case where there are two packets in each group with replace (APQ) is 206 Millions packets per second (Mp/s) and in the case of three packets per group, it is 150 Mp/s for a 256 queue capacity. In the case of the OPQ, the throughput is 122 and 99.5 Mp/s for the cases of two and three packets in each group of the PQs, respectively,

TABLE VI
RESOURCE UTILIZATION COMPARISON WITH KUMAR *et al.* [15], WITH 64-bit PDI

Queue Depth	Shift Register[a]		Systolic Array[a]		Hybrid P-Heap		OPQ Design ($N = 2$)	
	FFs	*LUTs*	*FFs*	*LUTs*	*FFs*	*LUTs*	*FFs*	*LUTs*
31	2077	4995	3999	8560	906	1411	2176	2494
63	4221	10275	8127	17520	1048	1996	4096	4719
127	8509	20835	NA	NA	1182	2561	8192	9362
255	NA	NA	NA	NA	1330	3161	16384	20003

[a] Shift register and systolic array architectures implementation are based on the work of Moon [9]

for similar queue capacity. Noting that the minimum number of operations required to pass a packet through the OPQ is two (enqueue and dequeue), in contrast to APQ which is one (replace). This achieved throughput is stable for the different queue capacities ranging from 34 to 1 Ki and for both queue types. Moreover, even in the worst case performance with $N = 64$, both OPQ and APQ can reach 40 Gb/s throughput for 84-B minimum size Ethernet packets, including minimum size packet of 64 B, preamble and interpacket gap of 20 B.

B. Comparison With Related Works

When compared to the systolic array in Moon's work [9] (see Table VI) supporting only enqueue and dequeue operations, our resource usage in terms of LUTs and FFs is lower. The resource usage results obtained with the proposed architecture (OPQ with 2 packets in each group) are comparable with the shift-register architecture in terms of FFs. They are lower in terms of LUTs (up to $N = 64$, our architecture remains comparable in terms of FFs), and they are higher than those reported for the hybrid p-heap architecture [15]. The reported design is entirely coded at high-level C++ language as compared to existing architectures coded at low-level in Verilog, VHDL, etc.

Table VII compares results obtained and reported in the literature with some relevant queue management architectures. The reported throughput of the QMRD [24] system depends on the protocol data unit payload size, while the reported

TABLE VII

MEMORY, SPEED, AND THROUGHPUT COMPARISON WITH QUEUE MANAGEMENT SYSTEMS, WITH 64-bit PDI

Queue Management System	Memory Hierarchy	BRAM Complexity	Performance	
			Speed (*MHz*)	Throughput (*Gb/s*)
OD-QM [3]	On-chip	56×36kbits	133	8
QMRD [24]	On-chip	389×18kbits	NA	< 9
NPMADE [25]	External SRAM	17×18kbits	125	6.2
APQ Design ($N = 2$)	NA	NA	201	103

OD-QM [3] results are for 512 active queues, and 64 bytes per packet. Our design is implemented with a total of 1-Ki queue PDIs capacity. The reported throughput is for the worst case egress port speed with 64-B sized packets, while offering 10× throughput improvements (APQ with $N = 2$). It should be noted that our design supports pipelined enqueue, dequeue and replace operations.

The number of cycles between successive dequeue–enqueue (hold) or replace operations, as depicted in Table VIII, for the OPQ is two clock cycles and only one clock cycle for the APQ supporting the replace. This is less than the FIFO [3] for 256-deep queues, binary heap [15], and p-heap architecture [16]. The reported shift register and systolic architectures in Moon's work [9] have both a latency of two clock cycles for en/dequeue. The systolic PQ described by Moon *et al.* [9] is not fully sorted until several cycles due to the fact that only one systolic cell is activated each time, i.e., the lower priority entry is passed to the neighboring block on the next clock cycle. In the case of the shift register proposed by Chandra and Sinnen [10], the performance degrades logarithmically. Compared to the p-heap architecture [16], even though it accept a pipelined operation each clock cycle (except in case of successive deletions), the latency is O(log(n)) in terms of the queue capacity against O(1) time latency for our proposed architecture.

For the PIFO queue [38], we implemented (placed and routed) the sorting data structure used in the PIFO block, called flow scheduler, on the ZC706 FPGA board with 16-bit priority and 32-bit metadata, using the Verilog code provided by the authors. This design supports a total capacity of 1024 elements. It is worth mentioning that this code was intended for a 16-nm standard cell ASIC implementation. Meanwhile, this architecture supports a dequeue–enqueue or replace each cycle. This architecture fully sorts all elements in parallel in a single pass through parallel comparators and encoder to determine the right position (the first 0–1 inversion) in which an incoming packet should be inserted. Both enqueue and dequeue operations require two clock cycles to complete. The FFs cost for PIFO is comparable to our architecture with a total of 58.5k FFs against 49.1k FFs, respectively. However, in terms of LUTs cost, our architecture (APQ with $N \leq 64$ and 32-bit metadata) is similar to the PIFO with 210 LUTs per element [see Fig. 8(b)] with 32-bit priority, while the cost in LUTs is only 149 per element for APQ with 16-bit priority. The total LUTs usage for the PIFO architecture is 215k LUTs. This expensive cost for the PIFO is mainly due to the extra logic necessary to fully sort the elements in the PIFO block, while our architecture partially sort the elements in each group, and it is capable to restore the queue invariants (see Section IV-C) in a single clock cycle.

TABLE VIII

PERFORMANCE COMPARISON FOR DIFFERENT PQ ARCHITECTURES

Architecture	Queue Depth	# Cycles per Hold Operation	Clock (ns)	Priority ; Metadata / Platform	Throughput (Mp/s)
FIFO PA-QM [3]	32	9 (no replace)	2.13 (64 queues)	NA ; 64 / FPGA (Virtex-5)	52.2
			2.16 (128 queues)		51.3
			2.67 (256 queues)		41.7
Shift Register [9]	64	2 (no replace)	13.5	16 ; NA / ASIC	37.0
	256		18.2		27.5
	1 Ki		20.0		25.0
Systolic Array [9]	64	2 (no replace)	20.8	16 ; NA / ASIC	24.0
	256		21.7		23.0
	1 Ki		22.2		22.5
Shift Register [10]	256	2 (no replace)	3.76	16 ; 32 / FPGA (Cyclone II)	133
	512		4.17		120
	1 Ki		4.92		102
	2 Ki		6.64		75.3
Hybrid Register/Binary Heap [13]	1 Ki	4 : replace 11 : others	~7.5	13 ; 51 / FPGA (Zynq-7)	Best: 33.3 Worst: 12.1
	2 Ki	2 : replace 11 : others	~8.0		Best: 62.5 Worst: 11.4
	4 Ki	1 : replace 11 : others	13.1		Best: 76.3 Worst: 6.9
	8 Ki	1 : replace 11 : others	15.0		Best: 66.7 Worst: 6.1
Pipelined Heap [16]	16 Ki	2 - 17 (no replace)	5.56	18 ; 14 / ASIC	Best: 90 Worst: 10.6
PIFO [38]	1 Ki	2: replace 4: others	13.9	16 ; 32 / FPGA (Zynq-7)	36.0 18.0
Proposed OPQ $N = 2$	1 Ki	2 (no replace)	3.31	16 ; 32 / FPGA (Zynq-7)	151
			4.14	32 ; 32 / FPGA (Zynq-7)	121
Proposed APQ $N = 2$		1: replace 2: others	4.0	16 ; 32 / FPGA (Zynq-7)	Best: 250 Worst: 125
		1: replace 2: others	4.98	32 ; 32 / FPGA (Zynq-7)	Best: 201 Worst: 100

Both architectures (OPQ and APQ) are capable of satisfying the invariants property for the entire queue in only one clock cycle (in each cycle all groups are being sorted in parallel). Also, this fixed number of cycles in our design is independent of queue depth unlike the O(log(n)) time for the dequeue operation with the heap [15], [26]–[28] and binary heap [13], where n is the number of nodes (keys). The achieved throughput is 151 Mp/s for the OPQ and 250 Mp/s for the APQ with 16-bit priority, and 32-bit metadata, against 76.3 Mp/s as the highest reported throughput in Table VIII for Huang's work [13], while

having the same FPGA board, Ioannou and Katevenis [16] with 90 Mp/s, and Chandra and Sinnen [10] with 102 Mp/s. The APQ is at least 2.45× faster than the latter architectures. For 32-bit priority with 32-bit metadata, our design achieves 121 and 201 Mp/s for OPQ and APQ, respectively. Moreover, the APQ is 2.0× faster than the reported works. Compared to [3], [9], [13], [16], [17], the reported throughput is independent of the queue depth.

Compared to existing NPU solutions like Broadcom [32], Mellanox (EZchip) NPS-400 [33], that can support up to 200 and 400 Gb/s, respectively, with built-in queue management systems, our proposed architecture is scalable in terms of performance for different queue capacities. Using a single FPGA (Zynq-7000), we can support links of 100 Gb/s with 64-B sized packets (32-bit priority with APQ). To scale up to 400 Gb/s and beyond, we can use a larger FPGA, for example, an UltraScale that has more logic resources could accommodate all design requirements, or using many FPGAs in parallel like in a multicard "pizza box" system, and/or some combination of these latters. Moreover, it should be noted that the FPGA solution is more flexible than the one of a fixed and rigid logic of an ASIC chip solution.

VII. Conclusion

This paper proposed and evaluated a priority queue in the context of flow-based networking in a TM. The proposed QM was entirely coded in C++, and synthesized using Vivado HLS. The resource usage of this implementation is similar to other priority queues in the literature, even though they were coded with low-level languages (Verilog, VHDL, etc.). Meanwhile, the achieved performance is at least 2× better than a comparable priority queue design, with a throughput of 10× faster than reported queue management system work in the literature for 1024 deep queues with 32-bit priority. Also, the achieved latency is in O(1) time for enqueue, dequeue, and replace operations, independent of the queue depth. HLS provides flexibility, rapid prototyping, and faster design space exploration in contrast to low-level hand-written HDL designs.

Future work will focus on integrating the proposed priority queue in a flow-based TM, and on assessing its capabilities and performance in practical high-speed networking systems.

VIII. Appendix: Summary of Notation

$\{\ \}$ Unordered set of elements.

$\langle\ \rangle$ Ordered vector of elements.

$\langle\{\ \}\rangle$ Elements of the set can be placed in any order in the vector.

$A \setminus B$ Set difference.

$\{\{X\}\} = \{X\}$, i.e., sets are flattened.

2nd max X = max $(X \setminus \{\max X\})$, i.e., the second largest element.

Acknowledgment

The authors would like to thank N. Bélanger, researcher at the École Polytechnique de Montréal, Montréal, QC, Canada, for his suggestions and technical guidance. They would also like to thank the anonymous reviewers for their valuable and enriching comments.

References

[1] N. Panwar, S. Sharma, and A. K. Singh, "A survey on 5G: The next generation of mobile communication," *Phys. Commun.*, vol. 18, pp. 64–84, Mar. 2016.

[2] S. O'Neil, R. F. Woods, A. J. Marshall, and Q. Zhang, "A scalable and programmable modular traffic manager architecture," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 2, May 2011, Art. no. 14.

[3] Q. Zhang, R. Woods, and A. Marshall, "An on-demand queue management architecture for a programmable traffic manager," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 10, pp. 1849–1862, Oct. 2012.

[4] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 351–362, 2011.

[5] Y. Xu, K. Li, J. Hu, and K. Li, "A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues," *Inf. Sci.*, vol. 270, pp. 255–287, Jun. 2014.

[6] C. Ni, C. Gan, and H. Chen, "Joint bandwidth allocation on dedicated and shared wavelengths for QoS support in multi-wavelength optical access network," *IET Commun.*, vol. 7, no. 16, pp. 1863–1870, Nov. 2013.

[7] S. H. S. Ariffin, J. A. Schormans, and A. H. I. Ma, "Application of the generalised ballot theorem for evaluation of performance in packet buffers with non-first in first out scheduling," *IET Commun.*, vol. 3, no. 6, pp. 933–944, Jun. 2009.

[8] R. Brown, "Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem," *Commun. ACM*, vol. 31, no. 10, pp. 1220–1227, Oct. 1988.

[9] S.-W. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Trans. Comput.*, vol. 49, no. 11, pp. 1215–1227, Nov. 2000.

[10] R. Chandra and O. Sinnen, "Improving application performance with hardware data structures," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum (IPDPSW)*, Apr. 2010, pp. 1–4.

[11] G. Bloom, G. Parmerl, B. Narahari, and R. Simha, "Shared hardware data structures for hard real-time systems," in *Proc. 10th ACM Int. Conf. Embedded Softw.*, 2012, pp. 133–142.

[12] P. Lavoie, D. Haccoun, and Y. Savaria, "A systolic architecture for fast stack sequential decoders," *IEEE Trans. Commun.*, vol. 42, no. 234, pp. 324–335, Feb. 1994.

[13] M. Huang, K. Lim, and J. Cong, "A scalable, high-performance customized priority queue," in *Proc. IEEE 24th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2014, pp. 1–4.

[14] C. N. G. Kumar *et al.*, "Improving system predictability and performance via hardware accelerated data structures," *Procedia Comput. Sci.*, vol. 9, pp. 1197–1205, Jun. 2012.

[15] C. N. G. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J Zambreno, and P. H. Jones, "Hardware-software architecture for priority queue management in real-time and embedded systems," *Int. J. Embedded Syst.*, vol. 6, no. 4, pp. 319–334, Sep. 2014.

[16] 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.

[17] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *Proc. 19th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, Mar. 2000, pp. 538–547.

[18] H. Wang and B. Lin, "Pipelined van Emde Boas tree: Algorithms, analysis, and applications," in *Proc. 26th IEEE Int. Conf. Comput. Commun.*, May 2007, pp. 2471–2475.

[19] R. Rönngren and R. Ayani, "A comparative study of parallel and sequential priority queue algorithms," *ACM Trans. Model. Comput. Simul. (TOMACS)*, vol. 7, no. 2, pp. 157–209, Apr. 1997.

[20] Center for Applied Internet Data Analysis. (Dec. 2017). *CAIDA Data—Overview of Datasets, Monitors, and Reports*. [Online]. Available: http://www.caida.org/data/passive/

[21] Y. Afek, A. Bremler-Barr, and L. Schiff, "Recursive design of hardware priority queues," *Comput. Netw.*, vol. 66, pp. 52–67, Jun. 2014.

[22] I. Benacer, F.-R. Boyer, N. Bélanger, and Y. Savaria, "A fast systolic priority queue architecture for a flow-based Traffic Manager," in *Proc. 14th IEEE Int. New Circuits Syst. Conf. (NEWCAS)*, Jun. 2016, pp. 1–4.

[23] I. Benacer, F.-R. Boyer, and Y. Savaria, "A high-speed traffic manager architecture for flow-based networking," in *Proc. 15th IEEE Int. New Circuits Syst. Conf. (NEWCAS)*, Jun. 2017, pp. 161–164.

[24] H. Fallside, "Queue manager reference design," Xilinx Inc., San Jose, CA, USA, Appl. Note 511, 2007.

[25] A. Nikologiannis, I. Papaefstathiou, G. Kornaros, and C. Kachris, "An FPGA-based queue management system for high speed networking devices," *Microprocess. Microsyst.*, vol. 28, nos. 5–6, pp. 223–236, Aug. 2004.
[26] X. Zhuang and S. Pande, "A scalable priority queue architecture for high speed network processing," in *Proc. 25th IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2006, pp. 1–12.
[27] K. McLaughlin, S. Sezer, H. Blume, X. Yang, F. Kupzog, and T. Noll, "A scalable packet sorting circuit for high-speed WFQ packet scheduling," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 7, pp. 781–791, Jul. 2008.
[28] H. Wang and B. Lin, "Succinct priority indexing structures for the management of large priority queues," in *Proc. 17th IEEE Int. Workshop Quality Service (IWQoS)*, Jul. 2009, pp. 1–5.
[29] D. E. Knuth, "The art of computer programming," *Sorting and Searching*, vol. 3, 2nd ed. Boston, MA, USA: Addison-Wesley, 1998, Sec. 5.3.4, pp. 219–247.
[30] T. Leighton, "Tight bounds on the complexity of parallel sorting," *IEEE Trans. Comput.*, vol. TC-34, no. 4, pp. 344–354, Apr. 1985.
[31] *10G Network Processor Chip Set (APP750NP and APP750TM)*, Agere Systems, Allentown, PA, USA, 2002.
[32] *200 G Integrated Packet Processor, Traffic Manager, and Fabric Interface Single-Chip Device*, document BCM88650, Broadcom, 2012.
[33] *NPS-400 400Gbps NPU for Smart Networks*, Mellanox (EZchip), San Jose, CA, USA, 2015.
[34] N. Possley, "Traffic management in Xilinx FPGAs," Xilinx, San Jose, CA, USA, White Paper WP244, 2006.
[35] "Enabling quality of service with customizable traffic managers," Altera, San Jose, CA, USA, White Paper WP-STXIITRFC-1.0, 2005.
[36] A. Khan *et al.*, "Design and development of the first single-chip full-duplex OC48 traffic manager and ATM SAR SoC," in *Proc. IEEE Conf. Custom Integr. Circuits*, Sep. 2003, pp. 35–38.
[37] B. Alleyne, "Chesapeake: A 50Gbps network processor and traffic manager," in *Proc. IEEE Hot Chips 19 Symp. (HCS)*, Stanford, CA, USA, Aug. 2007, pp. 1–10.
[38] A. Sivaraman *et al.*, "Programmable Packet Scheduling at Line Rate," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 44–57. [Online]. Available: https://github.com/programmable-scheduling/pifo-hardware/blob/master/src/rtl/design/pifo.v

Imad Benacer (S'16) received the B.E. degree in electrical and electronic engineering from Boumerdès University, Boumerdès, Algeria, in 2012 and the M.E. degree in electrical engineering from École Militaire Polytechnique, Algiers, Algeria, in 2014. He is currently working toward the Ph.D. degree at the École Polytechnique de Montréal, Montréal, QC, Canada.

His current research interests include the embedded implementation of image and video processing algorithms, network communication systems, and high-level synthesis targeting FPGA designs and implementations.

François-Raymond Boyer received the B.Sc. and Ph.D. degrees in computer science from the Université de Montréal, Montréal, QC, Canada, in 1996 and 2001, respectively.

Since 2001, he has been with the École Polytechnique de Montréal, Montréal, where he is currently a Professor at the Department of Computer and Software Engineering. He has authored or coauthored more than 30 conference and journal papers. His current research interests include microelectronics, performance optimization, parallelizing compilers, digital audio, and body motion capture.

Dr. Boyer is a member of the Regroupement Stratégique en Microélectronique du Québec, the Groupe de Recherche en Microélectronique et Microsystèmes, and the Observatoire Interdisciplinaire de Création et de Recherche en Musique.

Yvon Savaria (S'77–M'86–SM'97–F'08) received the B.Ing. and M.Sc.A. degrees in electrical engineering from the École Polytechnique de Montréal, Montréal, QC, Canada, in 1980 and 1982, respectively, and the Ph.D. degree in electrical engineering from McGill University, Montréal, QC, Canada, in 1985.

He has been a consultant or was sponsored for carrying research by Bombardier, CNRC, Design Workshop, DREO, Ericsson, Genesis, Gennum, Huawei, Hyperchip, ISR, Kaloom, LTRIM, Miranda, MiroTech, Nortel, Octasic, PMC-Sierra, Technocap, Thales, Tundra, and VXP. Since 1985, he has been with the École Polytechnique de Montréal, where he is currently a Professor at the Department of Electrical Engineering. He has out carried work in several areas related to microelectronic circuits and microsystems such as testing, verification, validation, clocking methods, defect and fault tolerance, the effects of radiation on electronics, high-speed interconnects and circuit design techniques, CAD methods, reconfigurable computing and applications of microelectronics to telecommunications, aerospace, image processing, video processing, radar signal processing, and digital signal processing acceleration. He has authored or coauthored 140 journal papers and 428 conference papers, and holds 16 patents, and he was the thesis advisor of 160 graduate students who completed their studies. He is currently involved in several projects that relate to aircraft embedded systems, radiation effects on electronics, asynchronous circuits design and test, green IT, wireless sensor network, virtual network, computational efficiency and application specific architecture design.

Dr. Savaria is a member of the Regroupement Stratégique en Microélectronique du Québec, the Ordre des Ingénieurs du Québec, and has been a member of the CMC Microsystems board since 1999 and he was the Chairman of that board from 2008 to 2010. He was a recipient of the 2001 Tier 1 Canada Research Chair on the design and architectures of advanced microelectronic systems that he held until 2015 and the 2006 Synergy Award of the Natural Sciences and Engineering Research Council of Canada. He was the Program Co-Chairman of ASAP'2006 and the General Co-Chair of ASAP'2007.