IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 19, 2019, accepted August 2, 2019, date of current version September 24, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2939154

HPQS: A Fast, High-Capacity, Hybrid Priority Queuing System for High-Speed Networking Devices

IMAD BENACER[1], FRANÇOIS-RAYMOND BOYER[1], AND YVON SAVARIA[2], (Fellow, IEEE)
[1]Department of Computer and Software Engineering, Polytechnique Montréal, Montréal, QC H3T 1J4, Canada
[2]Department of Electrical Engineering, Polytechnique Montréal, Montréal, QC H3T 1J4, Canada

Corresponding author: Imad Benacer (imad.benacer@polymtl.ca)

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, in part by Prompt Québec, in part by Ericsson Research Canada, in part by Mitacs, and in part by Kaloom.

ABSTRACT In this paper, we present a fast hybrid priority queue architecture intended for scheduling and prioritizing packets in a network data plane. Due to increasing traffic and tight requirements of high-speed networking devices, a high capacity priority queue, with constant latency and guaranteed performance is needed. We aim at reducing latency to best support the upcoming 5G wireless standards. The proposed hybrid priority queuing system (HPQS) enables pipelined queue operations with almost constant time complexity in practice. The proposed architecture is implemented in C++, and is synthesized with the Vivado High-Level Synthesis (HLS) tool. Two configurations are proposed. The first one is intended for scheduling with a multi-queuing system for which implementation results of 64 up to 512 independent queues are reported. The second configuration is intended for large capacity priority queues, that are placed and routed on a ZC706 board and a XCVU440-FLGB2377-3-E Xilinx FPGA supporting a total capacity of 1/2 million packet tags. The reported results are compared across a range of priority queue depths and performance metrics with existing approaches. The proposed HPQS supports links operating at 40 Gb/s.

INDEX TERMS Priority queue, networking devices, high-level synthesis, field-programmable gate array (FPGA).

I. INTRODUCTION

In modern routers, switches, line cards, etc., we find Network Processing Units (NPUs) [1]–[3]. They provide dedicated processing stages for traffic management and buffering. Traffic management includes policing, scheduling, shaping and queuing. For high-speed network switches and devices, queuing may represent a bottleneck. One of the feasible solutions to reduce queuing latencies is to tag the packets. This tagging will hold concise packet information for fast processing, while the actual packets are buffered independently by the NPU, thus reducing the queuing latencies between the different network processing stages [4].

Priority queues have been used in many applications such as event driven simulation [5], scheduling [6], real-time sorting [7], etc. A priority queue (PQ) can be represented as an abstract data structure that allows insertion and extraction of items in priority order. Different types of PQs have been proposed. In the literature, solutions span between the following: calendar queues [5], binary trees [8], shift registers [8]–[10], systolic arrays [8], [11], register-based arrays [12], and binary heaps [7], [12]–[14]. However, PQs can be divided in two classes: PQs with O(1) time complexity operations, independently of the queue size (number of nodes), and those with variable processing times.

One of the significant challenges facing network operators and Internet providers is the rising number of connected devices. This sets a need for scheduling, prioritizing packets of different applications, and routing the related traffic in a minimum time with the upcoming next generation cellular communication infrastructure (5G) [15]. Also, many applications must deal with real-time traffic, such as video streaming, voice over Internet protocol (VoIP), online gaming, etc. These applications require quality of service (QoS) guarantees. QoS are quantitative measures of the service provided by the network, for example, the average throughput, end-to-end delay,

The associate editor coordinating the review of this manuscript and approving it for publication was Cihun-Siyong Gong.

 This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

and packet loss. To provide such QoS for large numbers of connected devices, users, etc., high capacity priority queues must be used to maintain real-time sorting of queue elements at link speeds with guaranteed performance.

In this work, we propose a hybrid priority queuing system (HPQS) with two distinct configurations. The first configuration is intended for strict priority scheduling with distinct queues. The second configuration is a single high capacity queue intended for priority queuing. The HPQS would fit in the scheduling stage of today's data planes networking devices such as NPUs and switches (more details are given in Section II-A). Also, it can be used in different contexts such as traffic managers, task schedulers, sorting, etc.

This work is an extension of a previous related work [29]. The new contributions are as follows:

1) In the first configuration (distinct-queues model), we support full sort capability with the enqueue and dequeue operations. The second configuration (single-queue model) supports a third queue operation (replace). The HPQS supports independently all queue operations in a single clock cycle (see Section VI). The HPQS throughput can reach 40 Gb/s for minimum sized packets.
2) Analysis of HPQS operations leading to improvements that allowed matching the performance of hand-written register transfer logic (RTL) codes with high-level synthesis (HLS) design (see Section V). Moreover, by leveraging HLS, we have derived with a very small effort 92 substantially different implementations from the same high-level description (see Section VI-A) through model specialization.
3) Design space exploration targeting the ZC706 FPGA board and XCVU440 device was conducted. This exploration allowed characterizing resource usage (look-up tables and flip-flops), performance metrics (throughput, latency, and clock period), and power consumption of the HPQS design (more details are given in Section VI-A).

In this paper, we present placement and routing results of the HPQS in a ZC706 field programmable gate array (FPGA) board and XCVU440 device, the total capacity can reach 1/2 million packet tags of 16-bit priority keys in a single FPGA. The HPQS is proposed for high-speed networking devices operating in a constant 1-cycle latency per packet (queue operation) targeting 10 to 40 Gb/s network links. Moreover, the performance of the proposed HPQS is independent of the PQ capacity. Also, the proposed HPQS architecture is entirely coded in C++, providing easier implementation and more flexibility than some reported works in the literature [7], [8], [14], which use low-level coding, mostly in Verilog, VHDL, and targeting ASIC implementations.

The remainder of this paper is organized as follows. In Section II, we present a general switch architecture. That switch architecture provides a meaningful context where the proposed HQPS would fit. In Section III, we detail some related work on PQs found in the literature. In Section IV, the architecture of the proposed HPQS with its different configurations is presented. In Section V, we detail the HLS HPQS design methodology and considerations leading to the best performances. Section VI reports hardware implementation results and comparisons to related work. Finally, Section VII draws conclusions from this work.

II. BACKGROUND

In this section, we present the general architecture of a shared memory switch. Then, we elaborate on the concepts of scheduling and priority queuing.

A. NETWORK SWITCHES

Today's switches provide various sets of functionalities, from parsing, classification, scheduling and buffering of the network traffic. These functionalities can be supported by transformations applied to the traffic from the moment packets are received on input ports up to their transmission through destination ports. From the requirements of today's networks, switches must run at line rates of 10 to 40 Gb/s. The architecture of a shared memory switch, with its internal modules, is depicted in Fig. 1. An example of such switches in the literature is the Broadcom's Trident II series [16]. In these switches, a parser feeds packets from all ports into the ingress pipeline. After some processing in the match-action

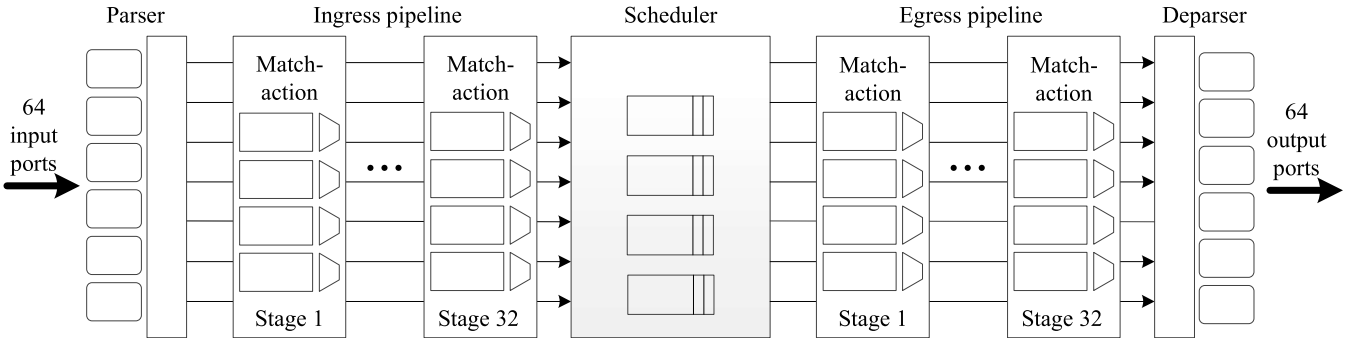

FIGURE 1. A general switch architecture with 64 input/output ports based on [17]. In this paper, we are mainly interested in the scheduler module.

stages, they enter a scheduler. Then, they exit from the egress pipeline [17]. Therefore, the switch must handle the aggregate processing requirements of all output ports at minimum packet size, such as 64 ports of 10 – 40 Gb/s each transmitting 84 byte packets (with preamble and interpacket gap). This translates into about 1 – 4 billion packets per second.

An HPQS is a powerful module that would fit in the scheduling stage of such switch architectures. The HPQS can be placed in each ingress port. A packet extracted from an HPQS would typically provide data to one egress port at a time. It will be shown that our proposed architecture is scalable in terms of performance for different queue capacities with network links of 10 – 40 Gb/s (see Section VI) using a single FPGA (Zynq-7000 and XCVU440 devices). To scale the implementation up to 64 ports, we can use many FPGAs in parallel like in a multicard ''pizza box'' system.

Means of scheduling and managing priorities with the considered class of architectures are explained in the next subsection.

B. SCHEDULING AND PRIORITY QUEUING

Network switches must provide scheduling capabilities (see Fig. 1). Some of the well-known scheduling algorithms are deficit round robin (DRR), fair queuing [18], and strict priority [1], etc. In this work, we are mainly interested in scheduling through strict priority and priority queuing, while providing large buffering capacity implemented using on-chip memories in an FPGA.

The high capacity priority queue is provided with guaranteed performance that can be used for sorting purposes. This sorting may represent the prioritization of the different class of service (CoS): voice, video, signaling, transactional data, network management, basic service, and low priority for each packet or flow. A flow maybe defined for example from the 5-tuple header information (source and destination IP, source and destination port, and protocol). The priority key is generated prior entry of the packet tag into the HPQS by the classification stage. It should be noted that packet classification is not discussed further in this paper, as we focus on strict priority scheduling and high capacity priority queuing with the proposed HPQS architecture in FPGA.

III. RELATED WORK

Several PQs have been proposed in the literature. These works can be classified as software-based and hardware-based solutions. In software-based solutions, we find mainly heaps and binary search trees [12], [19]. However, these implementations cannot handle large priority queues with high throughput and very low latency, due to the inherent $O(\log n)$ complexity per queue operation, where n is the number of keys.

Reported solutions for hardware PQs are based on calendar queues [5], binary trees [8], shift registers [8]–[10], systolic arrays [8], [11], and binary heaps [7], [12]–[14]. Moon [8] analyzed four scalable priority queue architectures based on: FIFOs, binary trees, shift registers and systolic arrays. Moon showed that the shift register architecture suffers from heavy bus loading, and that the systolic array overcomes this problem at the cost of doubling the hardware complexity. Meanwhile, the total capacity previously investigated by Moon is 1024 (1 Ki) elements. Also, the hardware approaches that were adopted limit queue size scalability due to limited resources. This motivated the research reported in the present paper to explore alternatives for building high capacity priority queues that can offer high throughput and low latency with O(1) time complexity (guaranteed performance). The reported solution is a hybrid PQ. Basically, hybrid PQs combine dedicated hardware approaches extended using on-chip or off-chip memories. In this work, we target to use only on-chip memories available in FPGAs (block RAMs).

Bhagwan and Lin [7] and Ioannou and Katevenis [14] proposed hybrid priority queue architectures based on a pipelined heap, i.e., a p-heap (which is similar to a binary heap). However, the proposed priority queue supports en/dequeue operations in $O(\log n)$ time against a fixed time for the systolic array and shift register, where n is the number of keys. Also, 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 sizes. The reported solutions implemented on ASICs had 64 Ki [14] and 128 Ki [7] as maximum queue capacities. Kumar [13] proposed a hybrid priority queue architecture based on a p-heap implemented on FPGA supporting 8 Ki elements. This architecture can handle size overflow from the hardware queue to the off-chip memory. Huang *et al.* [12] proposed an improvement to the binary heap architecture. Huang's hybrid PQ combines the best of register-based array and BRAM-tree architectures. It offers a performance close to 1 cycle per replace (simultaneous dequeue-enqueue) operation. In this solution, the total implemented queue capacity is 8 Ki elements when targeting the ZC706 FPGA board.

Zhuang and Pande [20] proposed a hybrid PQ system exploiting an SRAM-DRAM-FIFO queue using an input heap, a creation heap and an output heap. The packet priorities are kept in sorted FIFOs called SFIFO queues that are sorted in decreasing order from head to tail. The 3 heaps are built with SRAM, while the SFIFO queues extend the SRAM-based output heap to DRAM. Zhuang validated his proposal using a 0.13 μm technology under CACTI [21] targeting very large capacity and line rates: OC-768 and OC-3072 (40 and 160 Gb/s), while the total expected packet buffering capacity reached 100 million packets.

Chandra and Sinnen [9] proposed an extension of the shift register based PQ of Moon [8] using a software binary heap. For larger queue capacity implementation (up to 2 Ki), the resource consumption increases linearly, while the design frequency reduces logarithmically. This is a limitation for larger queues in terms of achieved performance and required hardware resources. Bloom *et al.* [10] proposed an exception-based mechanism used to move the data to secondary storage (memory) when the hardware PQ overflows.

Sivaraman *et al.* [17] 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 in a PIFO, manages to enqueue, dequeue, and replace in the correct order and in constant time a total capacity of 1 Ki elements.

McLaughlin [22], [23] proposed a packet sorting circuit based on a lookup tree (a trie). This architecture is composed of three main parts: the tree that performs the lookup function with 8 Ki capacity, the translation table which connects the tree to the third part, the tag storage memory. It was implemented as an ASIC using the UMC 130-nm standard cell technology, and the reported PQ had a packet buffering capacity of up to 30 million packet tags.

Wang and Lin [24], [25] proposed a succinct priority index in SRAM that can efficiently maintain a real-time sorting of priorities, coupled with a DRAM-based implementation of large packet buffers targeting 40 Gb/s line rate. This complex architecture was not implemented, it was intended for high-performance network processing applications such as advanced per-flow scheduling with QoS guarantee.

Afek [26] proposed a PQ using TCAM/SRAM. This author showed the efficiency of the proposed solution and its advantages over other ASIC designs [20], [22], [23], but its overall rate degrades almost linearly with larger queue size while targeting 100 Gb/s line rate. Also, Afek presented an estimation of performance with no actual implementation.

Van *et al.* [27] proposed a high throughput pipelined architecture for tag sorting targeting FPGA with 100 Gb/s line rate. This architecture is based on multi-bit tree and provides constant insert and delete operation requiring two clock cycles. The total supported number of packet tags is 8 Ki.

IV. THE HYBRID PRIORITY QUEUE ARCHITECTURE

In this section, we present the HPQS architecture. Then, we detail its different queuing models for scheduling and priority queuing, with the supported sorting types.

Fig. 2 depicts the proposed HPQS architecture. The HPQS assumes one input port representing the operation to perform and the pushed packet tag (Element In), and one output port representing the dequeued packet tag (Element Out). The HPQS is devised in four parts. The first part is the storage area of the queues implemented with on-chip Block RAMs (BRAMs). The second part contains the hardware PQ used to sort out packets in a single clock cycle, its depth represents the HPQS queuing width (QW), or simply the entire queue line. In this work, packets are sorted in ascending order of priority value based on the PQ presented in [28]. It should be noted that this work extends the PQ presented in [28] with full sort. The QW impacts directly the logic resources usage, especially in the hardware PQ. Moreover, the *number of queue lines* (QL) impacts the complexity of the exit buffer, and the priority encoder used in the push/pop indexes calculation module. Hence, to reduce the impact of QW and QL during the HPQS implementation (see Section VI), the HPQS

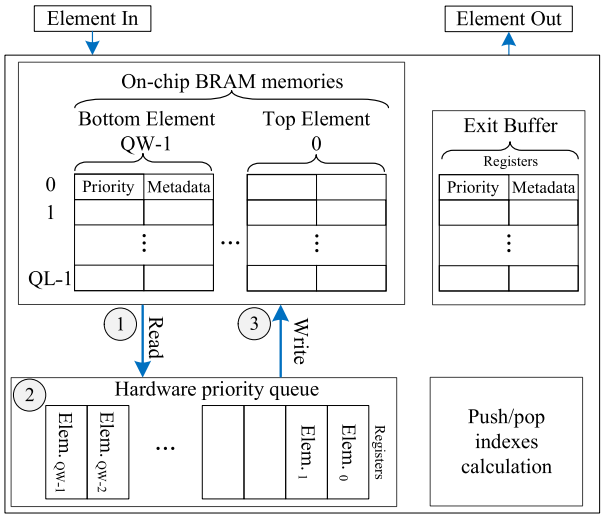

FIGURE 2. The proposed HPQS architecture.

has an almost square shape. Also, the HPQS is chosen for its guaranteed performance (almost constant time complexity in practice) per queue operation that is independent of its capacity. More details about the HPQS time complexity theoretical analysis are given in Section IV-C. The third part is represented by the exit buffer that is holding the top element for each queue line. The fourth part is the special module for push/pop indexes calculation.

From the same architecture of Fig. 2, we derive two queue models, where the main difference is how push/pop indexes are calculated. In type-1, each queue line is a distinct queue, with its own priority range, the first line having highest priorities, more details are given in Section IV-A. In type-2, all lines are used together as if they were a single large queue, more details are given in Section IV-B. The supported queue operations are enqueue and dequeue in type-1. A third operation, i.e., replace, is also supported in type-2. An enqueue enables insertion of an element to the HPQS, while a dequeue removes the highest priority element (lowest in priority value). The replace operation allows insertion while extracting the highest priority element. The whole HPQS design is described at high-level using the C++ language. The code is written in a way that allows efficient hardware implementation and prototyping in an FPGA platform. Moreover, this HPQS supports 512× with type-1, and 256× with type-2, larger capacity as compared to the PQ presented in [28] when targeting the same ZC706 FPGA board (see Section VI).

A. TYPE-1 DISTINCT-QUEUES MODEL

In the type-1 architecture, each queue line of the storage area represents a distinct queue. The system assumes the highest priority is the first queue line, while the lowest is the last one. Elements will be dequeued from first to last queue line according to their occupancy in ascending order of priority values. This represents a strict priority scheduler with distinct queues.

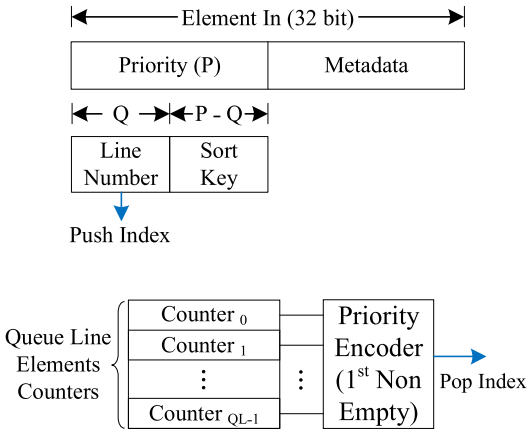

FIGURE 3. Push and Pop indices calculation in HPQS type-1 architecture.

When a packet tag is received (Element In) as depicted in Fig. 3, it contains its priority and required metadata information like the actual packet address, egress destination port, or any other attributes. The configuration of the data type limit can be modified for tag fields. The received packet tag is pushed with reduced priority information into the HPQS (priority information for sorting + push index) that is subsequently sorted in the appropriate queue. In this architecture, the priority key is 16-bit (P), and the word element length is 32-bit. The total number of queues, i.e. QL, is given by 2^Q.

Upon an enqueue operation, the line index is extracted from the priority of the received packet tag to select the line where this incoming element should be inserted. This is done by a simple bit extraction of the Q most significant bits, to have the push index. For a dequeue, a priority encoder finds the pop index of the highest priority element in the exit buffer by selecting the first non empty line from the HPQS line counters, see Fig. 3. Then, sorting in the hardware PQ, and storing the result back to the BRAMs are performed. The priority key that will reside inside each queue element can be further optimized to $P - Q + 1$ bits (the additional 1 bit is used to differentiate valid sort information from empty elements during hardware PQ sort). For example, for 64 queues, the priority key length stored inside each queue is only 11 bits. For 512 queues, the priority key length stored is 8 bits. Thus, the metadata field varies from 21 bits up to 24 bits for the same example (i.e., for 64 and 512 distinct queues, respectively). The on-chip memory BRAM_18K can hold a maximum of 512×32-bit elements per memory block. It will be shown that during implementation (see Section VI-A), the width of the HPQS is varied from 64 to 1024 (1 Ki) elements, while QL is varied from 64 to 512.

A counter per queue line is needed during pop index calculation for the design to be fully pipelined, as the HLS tool fails to meet the 1 cycle target while accessing directly the exit buffer due to carried dependency constraint. This carried dependency is between the store operation of the top element in the exit buffer after each HPQS operation, and the load operation of the top element in each line for pop index calculation. To prevent this dependency, we access the counters during pop index calculation, while the exit buffer is used to pass the output and to store top elements only. In addition, all counters contain the actual stored number of elements per queue line. When reaching or exceeding the queue line capacity during an enqueue, the specific queue line counter is halted, while the last queued element is dropped. Normal counter operation is resumed once a dequeue is performed. More details on how to achieve best performances and matching handwritten designs through HLS are given in Section V.

In this configuration, we propose two sorting types in the hardware PQ, a partial and full sort. Fig. 4 depicts the partial sort (called P. sort) architecture. The hardware PQ is divided in groups, a group contains two packets representing the min and max elements. Each group is being connected with its adjacent groups, and each independently applying in parallel a common operation on its data. This hardware PQ architecture is register-based single-instruction-multiple-data (SIMD), with only local data interconnects, and a short broadcasted instruction. More details are provided in [28].

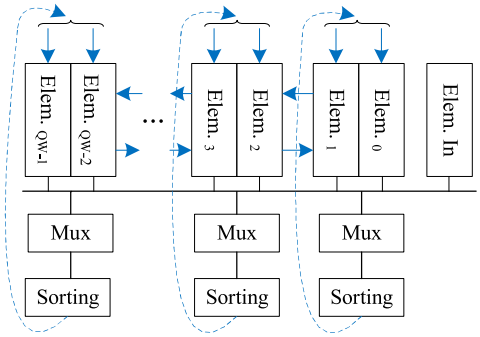

FIGURE 4. The hardware PQ architecture with partial sort.

Fig. 5 depicts the full sort (called F. sort) architecture. In here, the hardware PQ fully sorts the elements by comparing the incoming element to all existing elements during enqueue. The first activated comparator indicates the insertion location for the new element. This is found through a priority encoder leading to the appropriate enqueue index.

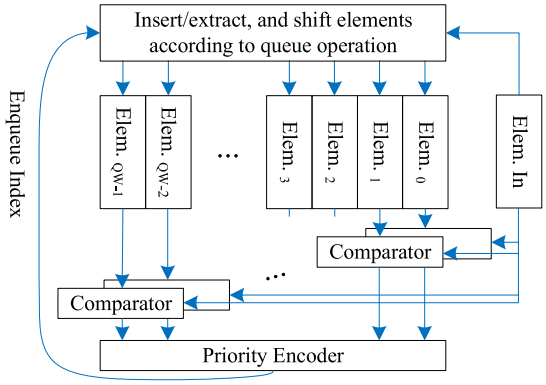

FIGURE 5. The hardware PQ architecture with full sort.

A shift register is used to move the elements beyond the enqueue index by one location to the bottom of the hardware PQ. During a dequeue, the top element is removed from the hardware PQ, while the remaining elements are shifted to the top by one location.

In partial sort, the elements with similar priorities are not distinguishable in their order of departure according to their order of arrival. Full sort is proposed to guarantee the order of departure for such packets, as the new incoming elements are enqueued and placed after the existing ones. Also, this is suitable for some network equipment where out of order reception is not supported.

B. TYPE-2 SINGLE-QUEUE MODEL

In this architecture, a high capacity PQ is proposed. In addition to enqueue and dequeue operations, a third basic operation called replace (simultaneous dequeue-enqueue) is supported. This architecture supports 16-bit priority and 48-bit metadata (64-bit elements) spread over distinct queue lines, similar to type-1, but virtually grouped to form a single queue. An enqueue is performed after receiving the line information of the first queue line with empty location through a priority encoder. This priority encoder selects the first non full line counter as depicted in Fig. 6. After sorting the upcoming element with the existing ones in the hardware PQ (load all elements from the BRAMs to the hardware PQ), the result is written to the same line in the BRAMs. In the case of a dequeue operation, a parallel comparison is made between the elements stored in the exit buffer with a binary parallel selection tree. The parallel selection tree has O(log QL) time complexity, this complexity is almost constant during implementation, see Section VI-A. The exit buffer holds the highest priority element of each queue line. From the corresponding pop index, the content of on-chip memories are sorted to complete the dequeue operation the same way an enqueue operation is performed. Also, for this configuration, it will be shown that during implementation, the width of the HPQS is varied from 64 to 1024 elements, while QL is varied from 64 to 512.

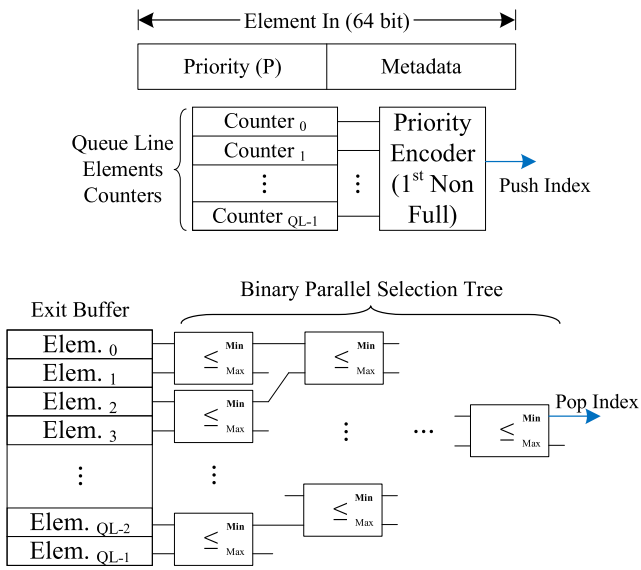

FIGURE 6. Push and Pop indices calculation in HPQS type-2 architecture.

In this HPQS configuration, even if full sort is used, we cannot guarantee the order of departure according to the order of arrival for similar priority elements from the different queue lines. The selection of the minimum element to dequeue is done by a binary parallel selection tree, while the push index is selected by a priority encoder from any queue line with empty location. As we can enqueue and dequeue from any queue line, the order of departure for similar priority elements cannot be guaranteed. In the type-1 architecture, this order is guaranteed by the hardware PQ (with full sort) and the exit buffer. Each queue line contains the elements in ascending order of departure, and the selection in the exit buffer is done by a priority encoder that will choose the best element from top to bottom in priority order. This guarantees the order of departure for similar priority elements. Therefore, in the type-2 architecture only partial sort is used.

C. HPQS TIME COMPLEXITY THEORETICAL ANALYSIS

The HPQS is composed of four parts (see Fig. 2), from which we can derive the time complexity.

1) *Storage area*: this storage exploits BRAMs, which can be accessed at each clock cycle through a read and a write operation (in dual port memories). From the HPQS BRAMs configuration (column wise), we can both load and store in pipelined fashion an entire line from the storage area to the hardware PQ, and vice versa, in only 1 clock cycle. It should be noted that with larger designs requiring more BRAMs, the routing resources impacts directly the read/write operation's propagation time. Meanwhile, our design has an almost constant time complexity in practice regardless of the number of BRAMs used, the delay being dominated by other HPQS modules (see Section VI-A).
2) *Hardware PQ with partial and full sort*: a PQ performing partial sort [28] has O(log N) time complexity, where N is the number of packet tags in each group, while the quality of dismissed elements when the queue is full is $1/N$ (lower is better). In this work, N is fixed to 2 packet tags in each group, for all queue depths. So, this lead to O(1) complexity during placement and routing of the partial sort in the FPGA. The full sort (explained in Section IV-A) is composed of parallel comparators that feed a priority encoder. The output of the priority encoder (insert location) is fed to a shift register. It should be noted that with larger QW, the slower is the priority encoder due to large fan-in. The time complexity of the full sort is almost constant time complexity in practice (see Section VI-A).
3) *Exit buffer*: this buffer contains a register-array that holds the top elements of each queue line. As we can load and store in the exit buffer at each clock cycle, the time complexity of this module is O(1).

4) *Push/pop indexes calculation module*: In the type-1 configuration, this module contains only queue line elements counters and a priority encoder (more details are given in Section IV-A). So, its time complexity is O(1). In the type-2 configuration, it contains in addition to the queue line elements counters and priority encoder, the binary selection tree. This selection tree has O(log QL) complexity. The QL needs to be small to lower the impact of this complexity. Therefore, we will show implementation results for QL varying from 64 up to 512. This leads to almost constant time complexity that in practice is essentially O(1). Also, the HPQS was chosen with an almost square shape to lower the logic resources usage, especially in the hardware PQ [28] and queue line elements counters (see Section VI).

D. HPQS FUNCTIONAL DESIGN

From a conceptual point of view, the HPQS is intended to work in a pipelined fashion. Each load/store from all BRAMs can be done in a single clock cycle. The HPQS operates as follows, in the first cycle (see circled numbers on Fig. 2), according to the operation to perform, either an enqueue or a dequeue in the type-1 architecture, in addition to replace in the type-2 architecture, an index is calculated. This index corresponds to the BRAMs line (together the respective lines of the parallel BRAMs contain 64 to 1 Ki elements, according to the queue capacity) to be loaded onto the hardware PQ. The ordering of the active queue is done in the second clock cycle, while a write back to the same BRAMs line to store the result is also done in the second clock cycle (More details are provided in Section V-A).

It should be noted that each BRAM holds up to 512 categories, and each category can have from 64 up to 1 Ki elements. This leads to 512 queues with at most 1 Ki capacity. The load and store operations require two distinct buses of 1024 × 32-bit to transfer the elements in type-1, and 1024 × 64-bit elements in type-2 architectures. They are necessary to transfer the stored elements in the BRAMs to the hardware PQ and vice versa. When the HQPS is generated with its full capacity, 2^{16} and 2^{17} nets are instantiated for each configuration, respectively. This impacts performance (more details are given in the implementation results Section VI).

V. HLS DESIGN METHODOLOGY AND CONSIDERATIONS

In this section, we first present the analysis of operations required by the proposed HPQS design. Then, we detail the steps applied in HLS to obtain the desired throughput and latency.

A. ANALYSIS OF HPQS OPERATIONS

The timing diagram demonstrating correct operation of the proposed HPQS is shown in Fig. 7. The required operations for the HPQS are to extract (in type-1 architecture) or choose (in type-2 architecture) the line to enqueue, dequeue, or replace an element, load the line content from the BRAMs to the hardware PQ for sorting, and finally write back the result to the same line in the BRAMs. Therefore, the HPQS operations consist in reading the storage memory (steps C0-C1), sorting the queue elements (step C1), and writing back the result to the same memory location (step C1). These are the specific tasks done by the proposed HPQS for each queue operation at any given clock cycle.

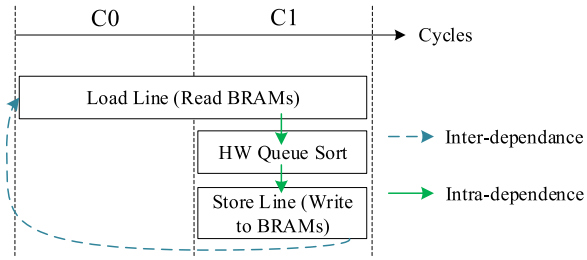

FIGURE 7. Proposed HPQS pipeline operations timing diagram.

B. DESIGN METHODOLOGY

1) HIGH-LEVEL DESIGN

High-level synthesis enables raising the design abstraction level, while providing more flexibility by automatically generating synthesizable Register Transfer Logic (RTL) from C/C++ models, as compared to Hardware Description Language (HDL) hand-written designs. Also, HLS requires less design effort, when performing a broad design space exploration, such that many derivative designs can be obtained with a small incremental effort. In this paper, we have derived 92 substantially different implementations from the same high-level description with HLS with a very small effort using the HPQS model parameters specialization such as the group size, QL, sort type (partial and full sort), HPQS configuration (type-1 and type-2), etc. This specialization is possible due to the elevated abstraction level at which the HPQS is expressed. In addition, design space exploration can be performed through different available directives and constraints provided by the tool. Using directives, a user can guide the HLS tool during C-synthesis. Thus, the designer can focus on the algorithmic design aspects, rather than on low-level details required when using HDL. HLS was chosen in this work for the above reasons.

The main metrics used to measure performance in HLS designs are area, latency, and Initiation Interval (II). In this work, we performed all experiments with Vivado HLS while the design is coded in C++. Appropriate design iterations were applied to refine the HPQS (code optimization and enhancement). The code was thoroughly tested.

The HPQS model is written in high-level C++0x. Indeed, Vivado HLS (version 2016.2) does not support yet C++11 or later. We used templated object oriented design for model specialization with the above mentioned parameters and data types lengths. Once we obtained a fully functional model, we proceeded to high-level synthesis, as explained in the following subsection.

2) HIGH-LEVEL SYNTHESIS

Prior HLS, design metrics are defined such as the target resource usage, desired throughput, clock frequency, and design latency. The RTL-equivalent (VHDL) of the C++ model is exported from Vivado HLS once it passes successfully the co-simulation with the high-level C++ HPQS design. Then, the Vivado tool is used to place and route the HPQS in the FPGA (more details are given in Section VI).

We performed a thorough design space exploration for the HPQS through HLS targeting minimum latency, equivalent memory usage (number of BRAMs) and highest throughput. The total considered HPQS capacity is 512 Ki. From Fig. 7, it can be seen that the minimum latency that can be achieved from our design operation, and initiation interval (II) are one clock cycle each, i.e., every clock cycle an output packet tag is ready. To target this optimal performance through HLS, the three directives that we focused on are: a latency directive targeting one clock cycle, a pipeline directive with II of one clock cycle, and a memory dependency directive asking for separate true dual port memories for accessing the element information through a read and/or write in the same cycle in the BRAMs. As HLS constraint, we target the lowest feasible clock period without violating the desired design latency and II mentioned above.

The partition directive was used to guide the tool to use only logic resources to implement logic and not the BRAMs available in the FPGA, to reduce design latency by cutting down the memory access time for the hardware PQ. To map the storage to the on-chip BRAMs, the resource directive is used with the option "true dual port RAM" enabling load/store in the same cycle. The pipeline directive is used to target an II of one clock cycle. All the mentioned directives are used together to generate the HPQS design. It should be noted that a bypass is not required for back-to-back similar line accesses in the BRAMs as the previous line content is already in the hardware PQ. In here, the load operation from the BRAMs is simply discarded (see intra-dependences in Fig. 7). More details on the experimental results of placement and routing in FPGA for different HPQS configurations and capacities are provided in Section VI-A.

VI. IMPLEMENTATION RESULTS

In this section, we detail the hardware implementation of our proposed HPQS architecture, resource usage and achieved performance, for different configurations (type-1 and type-2) and capacities (design scalability). Then, comparisons to existing works in the literature are discussed.

A. PLACEMENT AND ROUTING RESULTS

The proposed HPQS was implemented on a Xilinx Zynq-7000 ZC706 board (based on the xc7z045ffg900-2 FPGA) and on a XCVU440 Virtex UltraScale device (xcvu440-flgb2377-3-e), using Vivado tool with the *Explore* directive enabled. The type-1 architecture implementation results are summarized in Fig. 8, under ZC706 on the left column (Fig. 8a to 8d), and the XCVU440 results are summarized in the right column (Fig. 8e to 8h). Under ZC706 FPGA board, the resource utilization in terms of the number of look-up tables (LUTs) are reported in Fig. 8a, and of the number of flip-flops (FFs) in Fig. 8b. For performance, we report the achieved clock period in Fig. 8c. Also, the dynamic power consumption of the proposed HPQS type-1 architecture are depicted in Fig. 8d. In the same order, implementation results when targeting a XCVU440 device are reported in Fig. 8e–8h, respectively. In addition, the reported results are in terms of the HPQS QW, sort types (see the legends: partial sort labeled P. sort, and full sort labeled F. sort), and QL.

The type-2 architecture implementation results are reported in Fig. 9 in the same manner we reported the type-1 architecture implementation results. In here, the reported results are in terms of supported queue operations (with and without simultaneous dequeue-enqueue or replace), and the QL. Moreover, in the type-2 architecture, under ZC706 FPGA board, various HPQS queue widths are explored from 64 up to 512 elements. By contrast, the queue widths range from 64 to 1024 when targeting the XCVU440 device. These ranges are determined by the number of BRAM_18K available in the FPGA devices. The ZC706 FPGA board has only 1090 blocks, while the XCVU440 has 5040 blocks. Recall that a BRAM_18K can hold an entire column of the HPQS storage with 512×32-bit elements. In the type-2 architecture, each element is 64-bit that consumes 2 BRAMs to support the full width of each entry. It should be noted that the achieved throughput and latency for all HPQS configurations are one clock cycle.

In what follows, we discuss in details the obtained HPQS implementation results under both FPGA devices in terms of LUTs, FFs, achieved clock period and dynamic power consumption.

The resource consumption of the different HPQS configurations can be divided into four main parts: hardware PQ, exit buffer, storage resource, and counters of elements per queue line. The hardware PQ depth (QW) is varied from 64 to 1024 elements, while the maximum HPQS height is 512. In the hardware PQ implementation, only FFs and LUTs were used to obtain a fast pipelined architecture achieving one clock cycle per queue operation. The exit buffer was implemented as a register-based array to hold only the top elements of each queue line of the HPQS. The line counters are used to break up the dependency on checking if the queue line is empty/full at each queue operation on the exit buffer. Without these line counters, the HLS tool was not able to pipeline the design to one clock cycle. The BRAM_18K usage was found to reflect directly the width of the HPQS (QW), as each on-chip memory is mapped to hold an entire column of the proposed HPQS storage (see Fig. 2). The HPQS architecture for both configurations is scalable in terms of the number of BRAMs for 32 up to 1024 blocks as per the implementation results shown in Fig. 8 and Fig. 9.

In type-1 configuration (see Fig. 8a and 8e) with both FPGA devices (ZC706 and XCVU440, respectively),

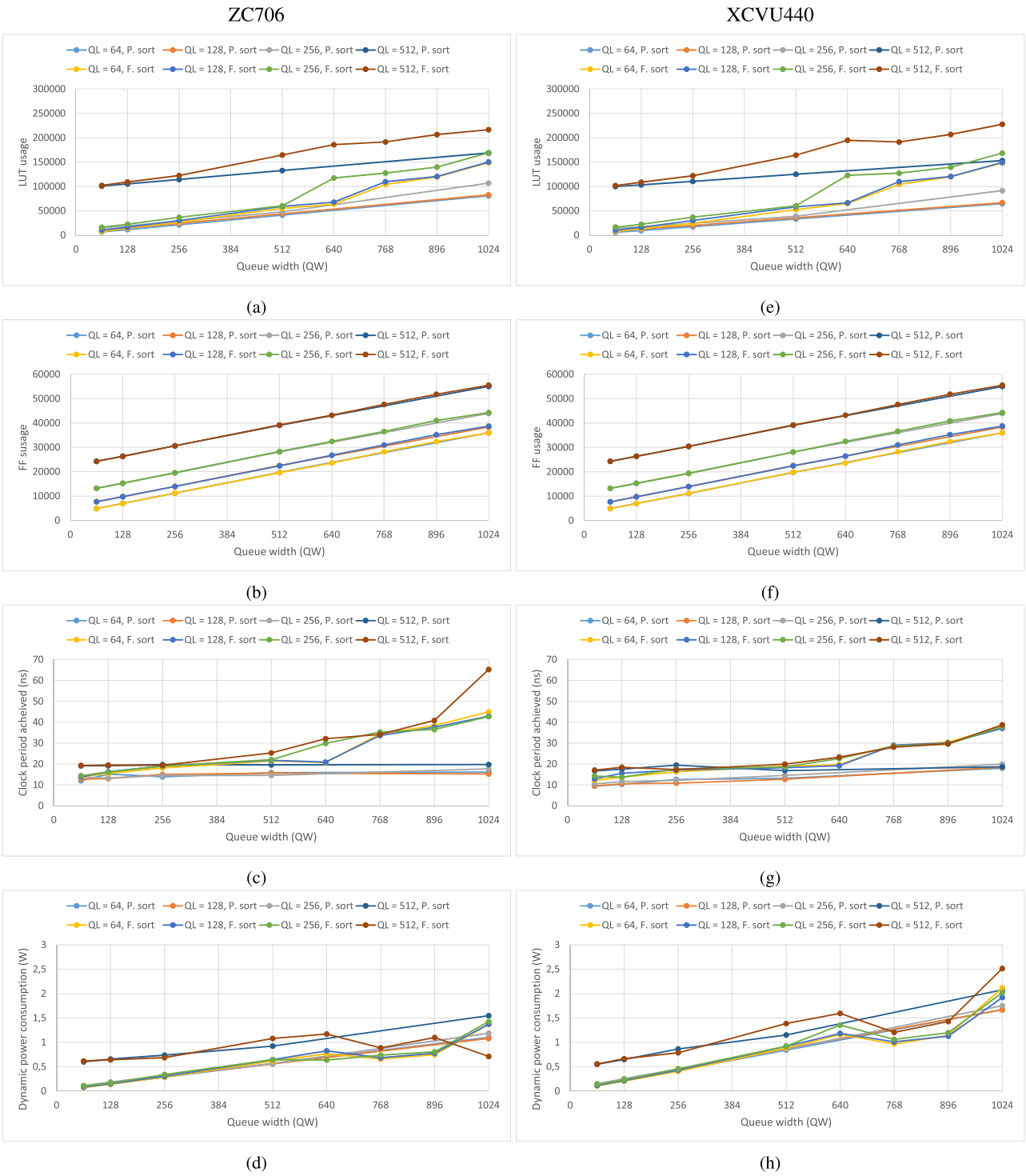

FIGURE 8. The HPQS type-1 configuration implementation results for 32-bit element on the ZC706 FPGA board, and XCVU440 device with: (a) LUT usage, (b) FF usage, (c) achieved clock period, and (d) dynamic power consumption using ZC706 FPGA board. Similarly from (e-h) under XCVU440 device.

increasing the capacity of the hardware PQ (QW) from 64 to 1024 elements with partial sort, the LUTs consumption linearly increases as more groups are attached in the SIMD hardware PQ. However, with full sort, this increase is linear over a range of capacity between 512-640 elements, then it stabilizes between 768-896 after which it increases again. This is the complexity of full sort with the priority encoder for index selection through an array of comparators (see Section IV-A). When increasing the number of queue lines or the height of HPQS, from 64 to 256 lines with its width not exceeding 512 elements, the LUTs usage is quite similar with both sort types (see Fig. 8a and 8e).

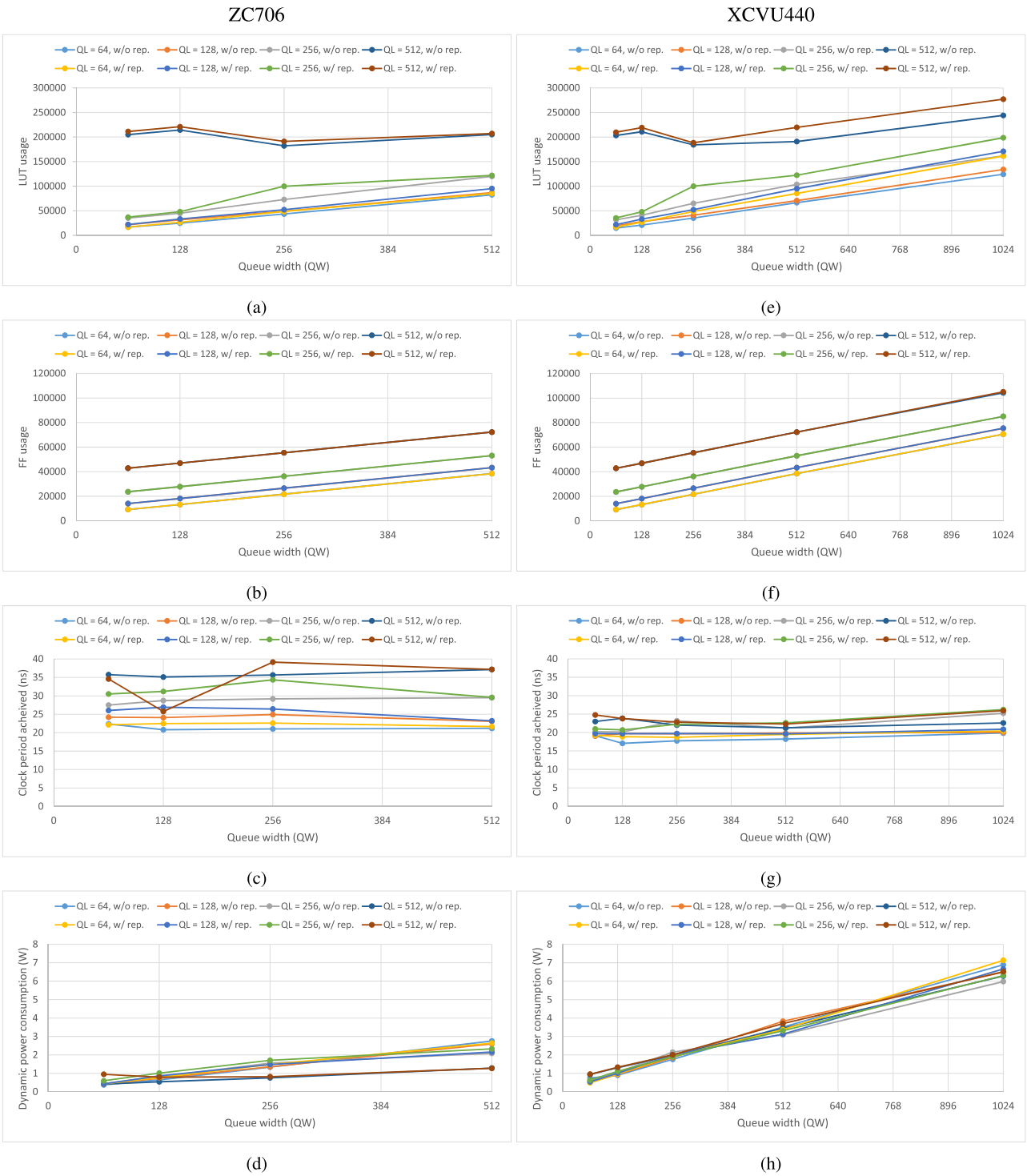

FIGURE 9. **The HPQS type-2 configuration implementation results for 64-bit element on the ZC706 FPGA board, and XCVU440 device with: (a) LUT usage, (b) FF usage, (c) achieved clock period, and (d) dynamic power consumption using ZC706 FPGA board. Similarly from (e-h) under XCVU440 device.**

Beyond 256 queue lines and 512 elements in HPQS width, the increase in the number of LUTs is no longer linear. The more lines in the HPQS, the more complex is the decoder of line index and its routing (width of the multiplexers), this complexity is logarithmic and appear to follow a stair case function. The same tendency can be seen for both devices (Zynq-7 and Virtex Ultrascale), where they have similar slice organization with 6-input LUTs. It should be noted that the XCVU440 device has more resources (11.5 × more in LUTs/FFs) compared to the ZC706 FPGA board.

Type-2 architecture has a selection tree used during dequeue, and a priority encoder for line selection during

enqueue in addition to the resource of type-1 architecture, with partial sort is used in hardware PQ. The LUTs usage (see Fig. 9a and 9e) is higher with similar tendency (linearly increasing for 64 up to 256 queue lines or HPQS height) and for different hardware PQ widths (64 up to 1024 elements). Also, supporting the third queue operation, i.e. replace, does increase the resource consumption by 37% in the worst case with the ZC706 FPGA board, and 54% with the XCVU440 device.

Regarding the FFs usage (see Fig. 8b, 8f for the type-1 architecture, and Fig. 9b, 9f for the type-2 architecture under ZC706 FPGA board and XCVU440 device, respectively), it reflects directly the use of memory resource of the hardware PQ, exit buffer and the line counters in all HPQS configurations. For example, the hardware PQ uses element length × queue width, for the counters 11-bit × queue lines, and the exit buffer element length × queue lines. let us recall that the element length in the type-1 architecture is 32-bit, while in the type-2 architecture is 64-bit. Also, as the HPQS capacity increases, all the above FFs resource relations are linear with the height and width dimensions.

For the performance metrics, the clock period achieved with the type-1 architecture (see Fig. 8c, 8g) with partial sort is better compared to the type-2 architecture (see Fig. 9c, 9g). It should be mentioned that the type-1 architecture is intended only for strict priority scheduling with distinct queues. If used as a priority queue, proper priorities repartition is advised. Type-2 is a high capacity priority queue that supports by default type-1 functionality. For type-1 with full sort, for both devices (see Fig. 8c, 8g), the performance decreases beyond 512 hardware PQ elements capacity with lower performances compared to partial sort. However, the achieved clock period is more stable and almost constant with partial sort in both FPGA devices. The XCVU440 device achieved better results than the Zynq-7. This is mainly due to the fact that the XCVU440 device is less prone to net congestion during routing, as it has more resources (11.5×) compared to the ZC706 FPGA board that used up to 99.0% of its LUTs, as reported in Table 1 for the largest design. It should be noted that for a clock period of 16.8 ns, the different designs are capable of supporting links up to 40 Gb/s for 84 bytes minimum size Ethernet packets (including minimum size packet of 64 bytes, preamble and interpacket gap of 20 bytes).

For dynamic power consumption, the XCVU440 device and ZC706 FPGA board have a similar tendency of linear growth with the HPQS capacity (for type-1 with partial sort and type-2 configurations, see Fig. 8d, 8h for the type-1 architecture, and Fig. 9d, 9h for the type-2 architecture under ZC706 FPGA board and XCVU440 device, respectively). It should be noted that with smaller designs, the lower the achieved clock period, the more power is consumed. This can be seen with designs having up to 256 lines with hardware PQ width up to 640 elements (in type-1 full sort, see Fig. 8d, 8h). Inversely, the larger is the design (beyond 256 queue lines), the higher is the clock period leading to lower power consumption. Overall, in type-1 configuration, both FPGA devices have similar tendency. With the type-2 architecture, the only difference is in the largest designs (HPQS height of 512, see Fig. 9d, 9h), the ZC706 FPGA board is nearly fully used leading to lower clock period and dynamic power consumption compared to the XCVU440 device.

Table 1 depicts the percentage of the FPGA resource usage, slice under ZC706 FPGA board, configurable logic block (CLB) under XCVU440 device, and BRAM memory after HPQS placement and routing of the largest designs (type-1 and type-2), with the achieved clock period, and dynamic power consumed in both FPGA devices. Note that in the ZC706 FPGA board, we used 88 to 99% of available slices, and 94.0% of BRAMs, that led to lower performance in comparison with the XCVU440 UltraScale device. The largest design (512 Ki capacity with type-2) consumes less than 20% of the CLBs and 40.6% of the BRAMs resources when targeting the XCVU440 device. So, the unused resources could be easily exploited to scale the HPQS design to even larger capacity beyond the proposed 512 Ki elements by 4.0×, and 2.0× for the type-1 and type-2 HPQS architectures with 1024 elements queue width, respectively. Note that when targeting the ZC706 FPGA board, the largest HPQS type-2 design implemented was 256 Ki elements in capacity due to limited number of BRAMs, leading to 94.0% memory usage.

TABLE 1. Percentage of resource utilization for the largest designs of HPQS in ZC706 and XCVU440 FPGAs.

Board / Device	HPQS Type	Design (Height × Width)	Slice/ CLB (%)	BRAM (%)	Clock Period (ns)	Dynamic Power (W)
ZC706 Zynq-7000	1	Partial sort 512×1024	88.4	94.0	19.75	1.55
	1	Full sort 512×1024	99.8	94.0	65.25	0.71
	2	w/o replace 512×512	98.6	94.0	37.13	1.28
	2	w/ replace 512×512	99.0	94.0	37.17	1.27
XCVU440 Virtex UltraScale	1	Partial sort 512×1024	9.2	20.3	18.62	2.08
	1	Full sort 512×1024	15.9	20.3	38.75	2.51
	2	w/o replace 512×1024	16.4	40.6	22.62	6.28
	2	w/ replace 512×1024	18.5	40.6	25.94	6.52

B. COMPARISON WITH RELATED WORKS

The proposed HPQS supports enqueue and dequeue operations for type-1, in addition to replace in type-2 configuration. The number of cycles between successive dequeue–enqueue (hold) operations is two clock cycles, and only 1 clock cycle when replace is supported, as reported in Table 2. Indeed, each queue operation takes one cycle to finish. This is less than the binary heap [12] and p-heap architectures [14]. The reported shift register and systolic architectures in Moon's work [8] have a latency of two clock cycles for en/dequeue. In case of the shift register proposed by Chandra and Sinnen [9], the performance degrades

TABLE 2. Performance comparison for different priority queue architectures.

Architecture	Queue Depth	# Cycles per Hold Operation	Clock (ns)	Priority ; Metadata / Platform	Throughput (Mpps)
Shift register [8]	1 Ki	2 (no replace)	20.0	16 ; NA / ASIC	25.0
Systolic array [8]	1 Ki	2 (no replace)	22.2	16 ; NA / ASIC	22.5
Shift register [9]	1 Ki 2 Ki	2 (no replace)	4.92 6.64	16 ; 32 / FPGA (Cyclone II)	102 75.3
Hybrid register/binary heap [12]	1 Ki	4 : replace 11 : others	~7.5	13 ; 51 / FPGA (Zynq-7)	33.3 12.1
	2 Ki	2 : replace 11 : others	~8.0		62.5 11.4
	4 Ki	1 : replace 11 : others	13.1		76.3 6.9
	8 Ki	1 : replace 11 : others	15.0		66.7 6.1
Pipelined Heap [14]	16 Ki	2 - 17 (no replace)	5.56	18 ; 14 / ASIC	Best: 90 Worst: 10.6
PIFO [17]	1 Ki	2 : replace 4 : others	13.9	16 ; 32 / FPGA (Zynq-7)	36.0 18.0
Lookup tree (trie) [22]	8 Ki	4: replace 8: others	7.0	NA ; 12 / ASIC	35.8 17.9
Multi-bit tree [27]	8 Ki	4 (no replace)	4.63	NA ; 12 / FPGA (Virtex II)	54
Hardware PQ [28]	1 Ki	2 : no replace 1 : replace	3.31 4.0	16 ; 32 / FPGA (Zynq-7)	151 250
Proposed HPQS Type-2	64 Ki	2 : no replace 1 : replace	23.08 23.22	16 ; 48 / FPGA (Zynq-7)	21.7 43.1
	256 Ki	2 : no replace 1 : replace	37.13 37.17		13.5 26.9
	64 Ki	2 : no replace 1 : replace	19.87 19.64	16 ; 48 / FPGA (XCVU440)	25.2 50.9
	512 Ki	2 : no replace 1 : replace	22.62 25.94		22.1 38.5

logarithmically. Compared to the p-heap architecture [14], even though it accepts pipelined operations each clock cycle (except in case of successive deletions), the latency is O(log n) in terms of the queue capacity, against constant time latency for our proposed HPQS architecture.

Even though the hardware PQ is fast, achieving 3.31 ns per operation with only enqueue/dequeue, and 4.0 ns with replace (see Table 2), the BRAMs distribution in the FPGA span many columns. During placement and routing of the largest HPQS designs (up to 512 lines $\times$ 1024 elements), long net delays tend to be generated by the hardware PQ to BRAMs connections (recall that the full architecture requires 2^{16} nets in the type-1 architecture and 2^{17} nets in the type-2 architecture to connect the BRAMs to the hardware PQ as explained in Section IV-D). This impacts directly the overall performance of the design as the clock period of the whole HPQS is 23.0 and 19.7 ns for the ZC706 and XCVU440 FPGA devices respectively with 64 Ki capacity (type-2). For 256 Ki capacity with the ZC706 FPGA board (largest routed design under this FPGA), it decreases to 37.0 ns. With 512 Ki design with XCVU440 device, the clock period decreases to 22.6 and 25.9 ns w/o and w/ support of replace operation, respectively. This design supports 1/2 million elements in a single FPGA, against a few thousands in previously published works.

In our proposed HPQS, we achieved a guaranteed performance and latency due to the fixed number of cycles (O(1) complexity). This constant number of cycles is independent of the hardware PQ width and the HPQS capacity, unlike the O(log n) time for the dequeue operation observed with the heap [12], [14], [20], where n is the number of nodes (keys). The throughput achieved with the proposed solution is 22.1 million packets per second (Mpps) without replace, and 38.5 Mpps with replace for 512 Ki total capacity (type-2) under XCVU440 device. From works depicted in Table 2, only some queues with 2 Ki and less capacity have throughputs better than our proposed HPQS. Beyond this capacity, either the designs have problem fitting in the targeted FPGA like in [8], [9], and [17], or the throughput degrades bellow our achieved throughputs [12], [14]. Compared to [4], [7], [8], [12], [26], the reported throughput of the HPQS is independent of the queue capacity.

VII. CONCLUSION

This paper proposed and evaluated a hybrid priority queue architecture intended to support the requirements of todays high-speed networking devices. The proposed HPQS was coded in C++ and synthesized using Vivado HLS. The first HPQS configuration with distinct-queues model is intended for strict priority scheduling. The second configuration is intended to offer a single large capacity priority queue for sorting purposes.

The proposed HPQS can support pipelined operations, with one operation completed at each clock cycle, with a capacity up to 1/2 million elements in a single FPGA. Also, the achieved throughput is comparable to similar related works in the literature, while supporting 10 to 40 Gb/s links. The achieved latency is in O(1) time complexity for the different queue operations independent to the total number of packet tags or HPQS capacity.

ACKNOWLEDGMENT

The authors would like to thank N. Bélanger, Researcher at Polytechnique Montréal for his suggestions and technical guidance and the anonymous reviewers for their valuable and enriching comments.

REFERENCES

[1] R. Giladi, *Network Processors: Architecture, Programming, and Implementation* (Systems on Silicon). San Mateo, CA, USA: Morgan Kaufmann, 2008.

[2] T. Wolf, "Challenges and applications for network-processor-based programmable routers," in *Proc. IEEE Sarnoff Symp.*, Princeton, NJ, USA, Mar. 2006, pp. 1–4.

[3] M. E. Kounavis, A. T. Campbell, S. T. Chou, and J. Vicente, "Programming the data path in network processor-based routers," *Softw., Pract. Exper.*, vol. 35, no. 11, pp. 1041–1078, 2005.

[4] 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.

[5] 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, 1988.

[6] 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.
[7] 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. Societies (INFOCOM)*, Mar. 2000, pp. 538–547.
[8] 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.
[9] 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.
[10] G. Bloom, G. Parmer, 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.
[11] 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.
[12] 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.
[13] 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.
[14] 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.
[15] 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.
[16] Broadcom. *High-Capacity StrataXGS Trident II Ethernet Switch Series*. Accessed: 2018. [Online]. Available: https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56850-series
[17] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown, "Programmable packet scheduling at line rate," in *Proc. SIGCOMM*, 2016, pp. 44–57.
[18] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Trans. Netw.*, vol. 4, no. 3, pp. 375–385, Jun. 1996.
[19] 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.
[20] 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.
[21] G. Reinmann and N. P. Jouppi, "CACTI2.0: An integrated cache timing and power model," Western Res. Lab., Univ. Avenue, Palo Alto, CA, USA, Res. Rep. 7, 2000.
[22] 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.
[23] K. McLaughlin, D. Burns, C. Toal, C. McKillen, and S. Sezer, "Fully hardware based WFQ architecture for high-speed QoS packet scheduling," *Integration*, vol. 45, no. 1, pp. 99–109, 2012.
[24] H. Wang and B. Lin, "Succinct priority indexing structures for the management of large priority queues," in *Proc. 17th IEEE Int. Workshop Qual. Service (IWQoS)*, Jul. 2009, pp. 1–5.
[25] H. Wang and B. Lin, "Per-flow queue management with succinct priority indexing structures for high speed packet scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 7, pp. 1380–1389, Jul. 2013.
[26] Y. Afek, A. Bremler-Barr, and L. Schiff, "Recursive design of hardware priority queues," *Comput. Netw.*, vol. 66, pp. 52–67, Jun. 2014.
[27] T. N. Van, V. T. Thien, S. N. Kim, N. P. Ngoc, and T. N. Huu, "A high throughput pipelined hardware architecture for tag sorting in packet fair queuing schedulers," in *Proc. Int. Conf. Commun., Manage. Telecommun. (ComManTel)*, 2015, pp. 41–45.
[28] I. Benacer, F.-R. Boyer, and Y. Savaria, "A fast, single-instruction–multiple-data, scalable priority queue," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 10, pp. 1939–1952, Oct. 2018.
[29] I. Benacer, F.-R. Boyer, and Y. Savaria, "HPQ: A high capacity hybrid priority queue architecture for high-speed network switches," in *Proc. 16th IEEE Int. New Circuits Syst. Conf. (NEWCAS)*, Jun. 2018, pp. 229–233.

IMAD BENACER received the B.E. degree in electrical and electronic engineering from Boumerdès University, Boumerdès, Algeria, in 2012, the M.E. degree in electrical engineering from École Militaire Polytechnique, Algiers, Algeria, in 2014, and the Ph.D. degree in computer engineering from Polytechnique Montréal, Montréal, QC, Canada, in 2019. His research interests include the embedded implementation of image and video processing algorithms, network communication systems, programmable data plane, software-defined networking, 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 Polytechnique Montréal, Montréal, where he is currently a Professor with the Department of Computer and Software Engineering. He has authored or coauthored more than 30 conference and journal articles. 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 Polytechnique Montréal, Montréal, Canada, in 1980 and 1982, respectively, and the Ph.D. degree in electrical engineering from McGill University, Montréal, 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 Polytechnique Montréal, where he is currently a Professor with the Department of Electrical Engineering. He has authored or coauthored 145 journal articles and 450 conference papers, and holds 16 patents. He was the Thesis Advisor for 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. His current research interests include 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 the applications of microelectronics to telecommunications, aerospace, image processing, video processing, radar signal processing, and digital signal processing acceleration.

Dr. Savaria has been a member of the CMC Microsystems Board, since 1999. He is a member of the Regroupement Stratégique en Microélectronique du Québec, the Ordre des Ingénieurs du Québec. 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. He was a Chairman of the CMC Microsystems Board, from 2008 to 2010.